PROTOTIPO DE MARCO DE TRABAJO PARA LA APLICACIÓN DE LA TEORÍA DE SISTEMAS COMPLEJOS ADAPTATIVOS EN LA ETAPA DE DISEÑO DE UN PROCESO DE DESARROLLO DE SOFTWARE

JOVANNY ANTONIO CASTAÑO MEJÍA

Trabajo de grado
Maestría

Director
Ana María López Echeverry
Magister en Ingeniería

UNIVERSIDAD TECNOLÓGICA DE PEREIRA
FACULTAD DE INGENIERIAS
INGENIERÍA DE SISTEMAS Y COMPUTACIÓN
PEREIRA
2018

TABLA DE CONTENIDO

TABLA DE ILUSTRACIONES

INTRODUCCIÓN

TÍTULO DEL PROYECTO

Prototipo[1] de marco de trabajo para la aplicación de la teoría de sistemas complejos adaptativos en la etapa de diseño de un proceso de desarrollo de software.

ANTECEDENTES Y MOTIVACIÓN

En su libro, "El quark y el jaguar. Aventuras en lo simple y en lo complejo", Murray Gell-Mann 1994, describe el concepto de Sistema Complejo Adaptativo, que en sus palabras reza: "un SCA es un sistema de ciclo cerrado (con realimentación), pero abierto al entorno, que realiza cambios en su estructura y comportamiento para alcanzar un objetivo"; lo que conlleva a la siguiente estructura/proceso:

Captura de datos - Monitoreo – Detección - Decisión – Actuación – Ejecutores

El objetivo de este proceso es identificar regularidades en el entorno del sistema mediante la captura de datos y convertirlas en acciones realizables por los ejecutores, respondiendo de esta forma a cambios instantáneos que se presentan en ese mundo exterior al que el sistema pertenece. De este proceso se espera emerja un comportamiento de adaptación que mejore la consecución de los objetivos del sistema. Los pasos intermedios pretenden separar las responsabilidades adecuadamente para alcanzar el objetivo de la adaptación, o alcanzar un estado deseable de un atributo descrito por un fenómeno emergente como totalidad (p.e orden, eficiencia, seguridad, etc.).

[1] Primer ejemplar que se fabrica de una figura, un invento u otra cosa, y que sirve de modelo para fabricar otras iguales.

El término complejidad se atribuye a volúmenes de interacciones altos entre entidades heterogéneas constitutivas de un sistema, un ejemplo de recurrente cita por autores del tema, es el flujo vehicular (Johnson, 2012.), en este libro el autor afirma: "Es posible captar la esencia de la complejidad mediante colecciones de objetos de toma de decisiones que pueden, o no, tener redes de conexiones entre ellos. " de esto, es necesario realizar una distinción entre mundos complicados y mundos complejos; un mundo complicado es lo que en la teoría general de sistemas se conoce como conglomerado (Bertoglio, 2004), la variedad de elementos que componen el sistema mantiene un grado de independencia de los otros, así, remover uno de estos elementos no fundamentalmente altera el comportamiento. La complejidad aparece cuando las dependencias entre los elementos llegan a ser importante, es decir, existe sinergia y dicha remoción impacta positiva o negativamente el comportamiento del sistema.

La acción de diseñar se enfoca en la toma de decisiones, buscando respuesta a las preguntas tipo: cómo, qué, cuándo y dónde. Es apropiado recordar las herramientas del diseño, entre estas se cuentan (Budgen, 2003): la separación de preocupaciones, la abstracción, el diseño modular y otro “conjunto de principios de ingeniería”, tales como, anticipación del cambio y diseño para generalidad.

En sí mismas, ellas proveen una guía al diseñador, sin embargo, es la experiencia el profesor magistral, que da la guía necesaria para el uso de dichas herramientas de forma efectiva. Ahora bien, el incremento en los niveles de heterogeneidad de los componentes de software, con frecuentes cambios en el contexto, metas y requerimientos durante el tiempo de ejecución, y las altas necesidades de seguridad y rendimiento, incrementa la incertidumbre sobre el diseño final, impactando de manera negativa la puesta a punto de la solución (Salehi Mazeiar and Ladan Tahvildari, 2009).

DESCRIPCIÓN DEL PROBLEMA DE INVESTIGACIÓN

El cambio es el endémico del software, se presenta en diferentes etapas del proceso de desarrollo de software, impactando de forma directa el diseño final del aplicativo. La característica que permite modificar el software para satisfacer requerimientos y circunstancias variantes se conoce como adaptabilidad, es claro, como los sistemas software se ven continuamente afectados por su entorno, entiéndase, hardware y software. Entre los muchos tipos de cambios están aquellos relacionados con el entorno de la aplicación, ante cambios inesperados el sistema software puede presentar problemas, debido en parte a que el diseño es “estático” en el sentido que no se diseña para todas los escenarios de operación posibles, si se emplea la adaptabilidad para flexibilizar su comportamiento como respuesta a diferentes condiciones externas, se podría, por lo menos en teoría, incitar la emergencia de fenómenos que impactan de forma positiva sus atributos de calidad. Esto sólo es posible en la medida que el software está diseñado para la adaptabilidad, pero ¿Se podría dar el diseño de un software basándose en el estudio de los Sistemas Complejos Adaptativos?, éste es el cuestionamiento que inicia esta aventura.

APROXIMACIÓN A LA SOLUCIÓN PARA UN SISTEMA DE CICLO CERRADO

La solución propuesta es la aplicación de teoría de sistemas complejos al diseño de un marco de trabajo para el desarrollo de sistemas software adaptables. Esto puede llegar a implicar:

- La caracterización de las propiedades de un sistema complejo adaptativo.

- El estudio del diseño de software mediante la revisión bibliográfica de las principales técnicas de diseño existentes y su afinidad con la adaptabilidad.
- La definición clara del ciclo de adaptación, definición de etapas y sus interfaces, buscando la separación de responsabilidades y la futura integración en un sistema software.
- Caracterización de patrones de diseño adaptables y propuesta integradora de dichos patrones.
- La propuesta de una arquitectura de un sistema software adaptable y del diseño para la adaptación.

MARCO TEORICO

Anteriormente, los sistemas de software se diseñaban para correr bajo entornos de ejecución con características muy limitadas; en los primeros programas era difícil diferenciar una estructura que facilitará el diseño modular y por ende la reutilización del código, en aquellas instancias, el empleo de lenguajes de programación de bajo nivel provocaba que los programadores emplearan gran cantidad de tiempo para garantizar el correcto funcionamiento de los programas codificados, así como su futuro mantenimiento. Como una estrategia para atenuar estas dificultades aparecen los *sistemas multitarea,* los cuales están bajo el control de un componente lógico de administración conocido como *el sistema operativo*, este incluye una capa de abstracción que oculta la complejidad asociada a la heterogeneidad de los componentes físicos de la máquina y que además sirve como interfaz lógica de administración de los recursos del sistema.

Un concepto introducido por este, *el proceso*, facilitaba dicha administración; los procesos son entidades software bajo un control centralizado orquestado por el kernel del sistema operativo, convirtiéndose en el mediador entre el programa y los recursos físicos del sistema.

La aparición de tales entidades lógicas mejora la adaptación a la demanda de recursos para la ejecución de diversas tareas, sin embargo, aumenta la dificultad de la administración, y complica la optimización del tiempo de utilización de los recursos. Para mantener la flexibilidad en la administración de las tareas, estas son encapsuladas dentro del proceso, estas entidades desconocen la complejidad computacional subyacente y por lo tanto no se puede garantizar un uso eficiente de los recursos.

Las ocasiones en que el software empieza a tener problemas, se presentan cuando algo inesperado cambia en el entorno. Es necesario revisar en las diferentes instancias de la etapa de diseño de un software, como incorporar características de adaptación ya sea en su propia arquitectura o como parte de su infraestructura subyacente, un ejemplo de esto es utilizar patrones de diseño que ayuda a resolver problemas complejos, o utilizar una nueva generación de middleware que provea el mecanismo central para crear sistemas software adaptativos.

El software adaptativo contendrá componentes que monitorearan su entorno (i.e. uso de CPU, capacidad de la red, y plataforma y uso de recursos de aplicación) y provee mecanismos de comunicación y administración para responder a cambios.

Diseño y arquitectura de software

La asunción típica del diseño de software es que el proceso puede proceder en la manera general de un diseño arquitectural o una ingeniería de diseño (Jones 1970), resumida en las siguientes cuatro etapas:

1. Etapa de factibilidad: identificar el conjunto de conceptos factibles.
2. Etapa preliminar de diseño: seleccionar y desarrollar el mejor concepto.

3. Etapa de diseño detallado: desarrollar las descripciones de ingeniería del concepto.
4. Etapa de planeación: evaluar y alterar el concepto para adaptarse a requerimientos de producción, distribución, consumo y retiro.

Cuando un proyecto da inicio, los analistas del sistema y los arquitectos de software interactúan con los *stakeholder* del proyecto; capturan los requerimientos del sistema, identificando sus restricciones y definiendo sus atributos de calidad, entonces, el arquitecto de software realiza una descomposición en funcionalidades y propone diseños con estilos factibles de arquitectura; cada diseño de arquitectura propuesto es mapeado con las funciones requeridas a los componentes identificados, una vez todas las funcionalidades requeridas son satisfechas se evalúan los atributos de calidad contra el diseño de arquitectura. Todo sistema, en su dominio de aplicación específico, puede tener requerimientos especiales para los atributos de calidad; cuando todos los atributos de calidad son representados cuantitativamente, el arquitecto de software puede enumerar diseños de arquitectura factibles y evaluar cada atributo de calidad para el diseño.

La arquitectura del software es el marco fundamental para estructurar el sistema. Las propiedades de un sistema tales como rendimiento, seguridad y disponibilidad están influenciadas por la arquitectura utilizada. En su libro “El proceso unificado de desarrollo”, Grady Booch, Ivar Jacobson, Jim Rumbaugh, 1999, exponen una aproximación de definición para Arquitectura de software que reza de la forma: “Un elefante es todo lo que el hombre ciego va encontrando – una gran serpiente (la trompa), un trozo de cuerda (la cola), un árbol pequeño (la pata)-.” De manera similar, la idea de arquitectura, al menos reducida a una sencilla frase definitoria, es lo que se encuentra en la mente del autor en ese punto del desarrollo. Para precisar esta definición, la arquitectura de software debe entenderse como un subconjunto de decisiones del cómo se

llevará a cabo la solución, de hecho, este subconjunto se caracteriza por reunir las decisiones más importantes sobre:

- La organización del sistema software
- Los elementos estructurales que compondrán el sistema y sus interfaces, junto con sus comportamientos, tal como se especifican en las colaboraciones entre esos elementos.
- La composición de elementos estructurales y del comportamiento en subsistemas progresivamente más grandes
- El estilo de arquitectura que guía esta organización: los elementos y sus interfaces, sus colaboraciones y su composición.

En relación con el último punto, los estilos arquitectónicos son diseñados para capturar el conocimiento de diseños efectivos en el alcance de metas específicas en un contexto de aplicación particular. Estos son definidos como una colección nombrada de decisiones de diseño arquitectural que: (1) son aplicables en un contexto de desarrollo dado, (2) restringen las decisiones de diseño especificadas a un sistema particular en ese contexto, y (3) obtiene cualidades beneficiosas en cada sistema resultante. La selección de los estilos arquitectónicos usualmente depende de la experiencia del arquitecto de y estos están habilitados para afectar de forma positiva o negativa diversos atributos de calidad.

La arquitectura de software está afectada no solo por la estructura y el comportamiento, sino también por el uso, la funcionalidad, un conjunto de atributos de calidad (como el rendimiento, la flexibilidad, la reutilización, la facilidad de comprensión), las restricciones y compromisos económicos y tecnológicos, y la estética

MARCO CONCEPTUAL

Adaptación: La sensibilidad al contexto se enlaza siempre al final con la etapa de adaptación. En este campo, la adaptación asiste a cambiar el comportamiento del sistema acorde a su entorno. Para tales sistemas la adaptación puede tomar dos formas. Adaptación estática que requiere reiniciar el sistema para ser implementado. Y la adaptación dinámica, que está habilitada para cambios en el comportamiento del sistema en tiempo de ejecución.

Arquitectura adaptativa: Responsable de identificar el conjunto de regularidades en el funcionamiento y entorno de los elementos estructurales y atributos de calidad y realizar modificaciones en los elementos estructurales.

Autonomic Computing: Nuevo paradigma para diseñar, desarrollar, implementar y administrar sistemas, inspirando las estrategias utilizadas por los sistemas biológicos para hacer frente a la complejidad, la heterogeneidad y la incertidumbre.

Esquema inicial: Es una estructura conceptual de la que el ser humano hace uso para comprender un conjunto de datos, e identificar fenómenos presentes en tal conjunto.

Feedback loop: Es un proceso en el que la información sobre el pasado o el presente influye sobre el mismo fenómeno en el presente o futuro. Como parte de una cadena de causa y efecto que forma un circuito o bucle, el evento es conocido como "feedback" en sí mismo.

Presiones selectivas: Fuerzas que actúan sobre las poblaciones que determinan que algunos individuos son más estructuralmente adecuados que otros y contribuyen más a las generaciones posteriores.

Sensibilidad al contexto: En informática sensibilidad de contexto se refiere a la idea de que las computadoras pueden tanto sentir, y reaccionar en función de su entorno. Las entidades físicas o lógicas pueden tener información sobre las circunstancias en las que operan y basado en reglas, o un estímulo inteligente, reaccionar en consecuencia.

ESTADO DEL ARTE

Según [Salehie y Tahvildari, 2005] el proceso de adaptación del software es dependiente de las propiedades de adaptación conocidas como propiedades auto-*, además de las características del dominio como modelos o información del contexto, y las preferencias de los stakeholder. Un referente fundamental en el estudio del proceso de adaptación tiene que ver con la línea de computación autonómica, [Kephart y Chess 2002] hace referencia a sistemas que pueden ser autoadministrados dado un conjunto de objetivos de alto nivel, especificados por administradores del sistema, una idea introducida en el 2001 en la National Academic of Engineers at Harvard University, por Paul Horn, el entonces Vicepresidente Senior de Investigación de IBM.

La esencia de la computación autonómica es la autoadministración, es decir, que el sistema mantenga y ajuste su operación de cara a cambios en los componentes, cargas de trabajo, demandas, y condiciones externas. Cuatro propiedades de la auto-administración bajo el enfoque de la computación autonómica, conocidas también como propiedades auto-* son: la auto-configuración de componentes y subsistemas mediante la directiva de una política de alto nivel; la auto-optimización, los componentes y subsistemas continuamente buscan oportunidades para mejorar su performance y eficiencia; la auto-recuperación, el sistema detecta automáticamente, diagnostica y repara problemas localizados; y finalmente la auto-protección, donde el sistema automáticamente se defiende contra ataques maliciosos y fallas.

Una aproximación más completa a la caracterización de la computación autonómica la realiza IBM Zurich Research Laboratory [Koehler, Giblin, Gantenbein y Hauser 2003], que las características mencionadas por Kephart adicionan; la identidad del sistema, el sistema tiene conocimiento de sus componentes o estado actual, funciones e interacciones; la auto-adaptación, el sistema busca maneras de mejorar la interacción con los sistemas vecinos describiéndose para otros y descubriendo otros sistemas en el entorno; finalmente un sistema autonómico oculta la complejidad subyacente de las tareas a sus usuarios.

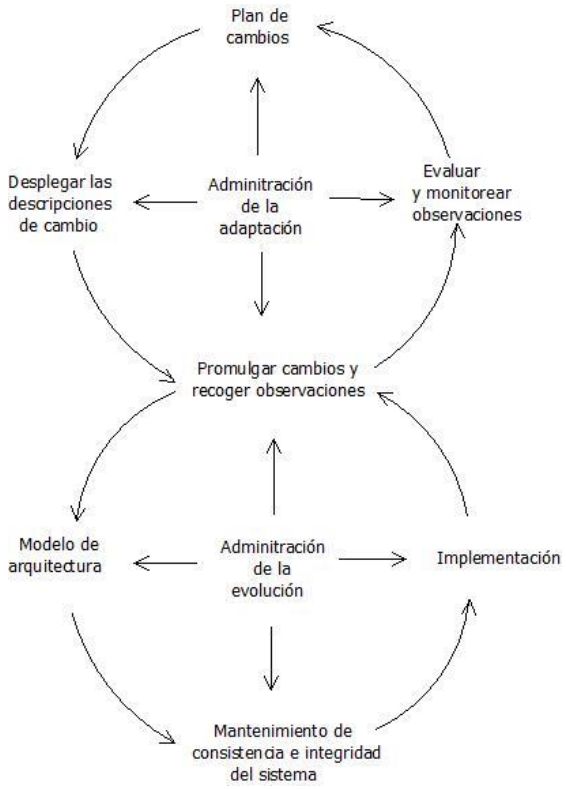
Nuevamente [Salehie y Tahvildari, 2005] completan las características autonómicas especificando las propiedades de auto-sensibilidad, donde un sistema autónomo es sensible a su estado y a su comportamiento para autoadministración y colaboración con otros sistemas; además de la propiedad de sensibilidad al contexto donde un sistema autónomo debe ser sensible a su entorno de ejecución y reaccionar a cambios en él; y finalmente la anticipación, es la estimación de los recursos necesarios para llevar a cabo las tareas ocultando su complejidad.

Según [McKinley, Sadjadi, Kasten y Cheng 2004], se afirma que la computación adaptativa toma fuerza con el desarrollo de dos campos emergentes la computación ubicua, que se enfoca en disolver los límites tradicionales del cómo, cuándo y dónde humanos y computadores interactúan, y segundo el crecimiento de demanda de computación autonómica, que como se observó se enfoca en el desarrollo de sistemas que puedan gestionarse y protegerse por sí mismos. En esta revisión se describen también dos aproximaciones para implementar software adaptativo: la primera de ellas conocida por la utilización de parámetros adaptativos, consiste en modificar las variables del programa que determinan su comportamiento, para de esta forma, guiar a una aplicación en el uso de una estrategia diferente existente,

pero no adopta nuevas estrategias, tiene el inconveniente de no permitir que nuevos algoritmos y componentes sean adicionados a la aplicación después de su diseño y construcción original. Por otro lado, la aproximación por adaptación de composición permite cambiar algoritmos o componentes estructurales del sistema por otros que mejoren o ajusten el programa a su entorno o condiciones internas.

Lo anterior se entiende mejor desde el contexto de la noción de adaptación y su íntima relación con la noción de evolución, [Oreizy y Gorlick 1999] presenta el modelo de la Figura,

Figura 1. Procesos de alto nivel, aproximaciones de propósito general a los sistemas auto-adaptativos



Fuente. An architecture-based approach to Self-Adaptive software

En el modelo de la Figura 1, un sistema auto-adaptativo observa su propio comportamiento y analiza estas observaciones para determinar adaptaciones

apropiadas, es lo que se muestra en la parte superior del ciclo donde se realiza una recolección de observaciones para su posterior monitoreo y evaluación de donde se espera poder planear cambios cuyas descripciones puedan ser desplegadas, esto se conoce como administración de la adaptación.

El modelo de administración de la evolución se refiere a aquellos cambios hechos en un punto neurálgico del sistema software, como lo es su arquitectura, los cambios sobre esta pueden incluir adición, remoción, reemplazo de componentes y conectores, modificaciones para la configuración o parámetros de componentes o conectores, y alteraciones en las redes de topologías de componentes.

Esta primera separación descrita por ambos modelos, se integran bajo el concepto de adaptación, incluso muchos trabajos posteriores incluyen los temas de arquitectura en el proceso adaptativo.

[Magee, Rogério, Giese, Inverardi 2008] describe las líneas principales de investigación para la ingeniería de software de sistemas auto-adaptativos, definen las dimensiones de modelado como los puntos de variación de los sistemas auto-adaptativos, donde cada dimensión describe un aspecto particular del sistema, los cuatro grupos de dimensiones son: metas del sistema, causas de la auto-adaptación, mecanismos de auto-adaptación y efectos sobre el sistema. Algunos retos de investigación para cada grupo de dimensiones de modelado: para las metas del sistema, ¿cómo resolver conflictos entre metas?; para las causas, ¿cómo compensar el sobrecosto asociado a monitoreo del sistema cuando se tienen diferentes propiedades de QoS?; en los mecanismos, ¿cómo manejar la simultaneidad de las tácticas de adaptación?, ¿cómo acomodar una aproximación de ingeniería sistemática que integre aproximaciones de ciclo de control y aproximaciones relacionadas a la adaptación?; finalmente en los efectos, ¿cómo predecir el comportamiento exacto de un software debido a cambios en tiempo de ejecución?

[Salehie and Ladan Tahvildari 2009] identifican dos categorías de aproximaciones al diseño de sistemas de computación autonómica: diseños fuertemente acoplados y diseños desacoplados. El primero es la forma más frecuente, consiste en construir agentes inteligentes con sus propias metas, y el segundo es aquel en el cual la infraestructura subyacente habilita el comportamiento autonómico del sistema. Ambas aproximaciones requieren dos tipos de tareas: i) implementar funcionalidades en el sistema objetivo y ii) introducir algunos patrones para autoadministración. Este conjunto de elementos abarca dos niveles de la arquitectura: relaciones intra-elementos y relaciones inter-elementos.

En el aspecto específico de auto adaptación, se han presentado diversas técnicas y formalismos para abordar su estudio.

La computación reflexiva [McKinley, Sadjadi, Kasten y Cheng, 2004] se refiere a la habilidad del programa para razonar acerca de su comportamiento (introspección) y la posibilidad de alterar su comportamiento en función de sus observaciones (intercesión). Una tecnología asociada con esta habilidad es MOP, meta-object-protocol es una interface que habilita de forma sistemática la introspección y la intercesión a un nivel basado en objetos.

El diseño basado en componentes soporta dos tipos de composición, composición estática donde se combinan varios componentes en tiempo de compilación para generar una aplicación, y composición dinámica donde se puede adicionar, remover, o reconfigurar componentes en una aplicación en tiempo de ejecución. Las tecnologías que habilitan este aspecto, básicamente se relacionan con el middleware que provee una capa que los desarrolladores pueden explotar para implementar comportamiento adaptativo.

[Kephart y Chess 2002], también presentan algunas consideraciones a nivel arquitectónico, dado que el sistema tendrá conjuntos de elementos autónomos en continua interacción, se propone la estructura de un elemento autonómico, este modelo se ajusta perfectamente al ciclo de adaptación descrito en este documento, la premisa es que cada elemento autonómico debe ser responsable por administrar su propio estado interno, comportamiento y sus interacciones con el entorno. Hacen una aproximación de un listado de retos de ingeniería relacionados con la temática: el ciclo de vida de un elemento autonómico, relaciones entre elementos autonómicos, especificación de metas, por mencionar algunos. Además de algunos retos científicos como son: modelos y abstracciones de comportamiento, teoría de robustez, teoría de optimización y aprendizaje, teoría de negociación y modelado estadístico automatizado.

Un aporte interesante de este artículo es la aproximación a una arquitectura genérica para sistemas autonómicos, en primera instancia dentro de un entorno contextual se encuentra una base de conocimiento con un estilo arquitectónico similar al blackboard, utilizado por los siguientes componentes lógicos: negociación, ejecución y observación. El componente de negociación mantiene intercambio con el entorno, negocia el cumplimiento de los requerimientos

solicitados y comunica sus propios requerimientos a otros sistemas, el propósito de este componente es la especificación del comportamiento meta. El componente de ejecución se concentra solo en llevar a cabo las acciones relacionadas con el comportamiento en un entorno específico. Finalmente, el componente de observación recibe el estado de información del entorno, observa los efectos de lo que el componente de ejecución está realizando, sin conocimiento de lo que actualmente fue ejecutado, adiciona sus observaciones al conocimiento compartido, y produce una representación de sus observaciones para análisis.

Años atrás [Blair, Coulson, Robin y Papathomas 1999] describen las así llamadas arquitecturas para middleware de próxima generación, en su artículo proponían el diseño de middleware configurable, utilizando estándares abiertos y basado en el concepto de reflexión, introducen una arquitectura independiente del lenguaje de programación usando un metamodelo para estructurar meta-espacios y utilizando grafos de objetos para realizar composición de componentes. Como es sabido, el middleware emerge como un componente de la arquitectura importante para aplicaciones distribuidas, su rol es presentar un modelo unificado de programación que abstraiga al desarrollador de temas relacionados con la heterogeneidad en sistemas distribuidos. La idea básica de la reflexión es que al exponer la implementación subyacente llegue a ser posible insertar comportamiento adicional para monitorear dicha implementación e inclusive puede llegar a ser usado para adaptar el comportamiento interno del sistema.

Esta aproximación adopta un modelo de computación orientada a objetos dada la predominancia en modelos de procesamiento distribuido, caracterizando el modelo de la siguiente manera: i) los objetos pueden tener múltiples interfaces, ii) las interfaces deben ser soportar de flujo y señales y iii) puede haber enlace explícito entre interfaces compatibles. La propuesta mantiene meta-espacios por objeto y soporta una aproximación procedimental para la reflexión.

[Magee y Kraner 1998] tienen una aproximación distinta para la adaptación, basada en ADL's, lenguajes de descripción de arquitectura, que permiten la modificación dinámica de la arquitectura de software. Los lenguajes de descripción de arquitecturas son notaciones para representar diseños arquitecturales y estilos arquitectónicos, con los años han evolucionado hacia configuraciones de sistemas distribuidos complejos, tienen como característica permitir la descripción de estructuras dinámicas que evolucionan con el progreso de la ejecución, involucrando cambios en los componentes y

conexiones. Este artículo se centra en la descripción de DARWIN, un ADL que permite a un programa distribuido ser especificado como una construcción jerárquica de componentes, dichos componentes interactúan accediendo a servicios. Estos componentes son vistos en términos de los servicios que proveen y los servicios que requieren; la ubicación de cada componente es transparente para el resto, produciendo independencia del contexto, el propósito de este ADL es construir tipos de componente básicos o compuestos declarando instancias de estos y enlaces entre los servicios requeridos y los proveídos por otros.

Por su parte [White, Hanson, Whalley, Chess y Kepharn], motiva una aproximación de arquitectura para computación autonómica, en esta aproximación todo componente del sistema es un elemento autonómico, dichos elementos incluyen recursos de computación tales como bases de datos, sistemas de almacenamiento y servidores. Para dicha aproximación se conservan un par de principios; el primero de ellos es que no se debe imponer ningún requerimiento sobre la estructura interna de los componentes, solo la descripción de las interfaces externas y comportamientos requeridos; el otro principio es que el sistema debe ser descrito como la agregación de componentes autonómicos con el fin de que el sistema como un todo, presente propiedades auto*.

El concepto clave de esta arquitectura es el de elemento autonómico siendo aquel responsable de administrar su propio comportamiento regulado por políticas a alto nivel y acorde con la relación con otros elementos de este tipo. Un elemento autonómico debería cumplir los siguientes comportamientos, ser autoadministrado, tener capacidad para establecer y mantener relaciones con otros elementos autonómicos, administrar su comportamiento y relaciones, ajustarse a requerimientos realistas cuando necesita un servicio de otro elemento, ofrecer rangos de calidad apropiados para propiedades no funcionales, brindar autoprotección contra solicitudes inapropiadas.

Finalmente [Morin, Hassine, Jézéquel, y Barais, 2009] abordan el tema de unificación de la adaptación y la evolución del diseño desde una perspectiva diferente a lecturas anteriores, por un lado, aborda la auto-adaptación con el objetivo de ajustar la calidad del servicio, y por otro lado estudia las decisiones de evolución del diseño para asegurar alta disponibilidad. Se aclara que la evolución del diseño y la auto-adaptación no son independientes, y reflejar una evolución de diseño sobre un sistema auto-adaptativo en tiempo de ejecución no siempre es seguro. El artículo propone unificar adaptación y evolución en tiempo de ejecución, monitoreando tanto la plataforma en tiempo de ejecución y el modelo del diseño original. Correlacionar esos eventos heterogéneos y usar pareo de patrones sobre eventos para tomar una decisión pertinente para la adaptación en tiempo de ejecución.

El estudio en esta y áreas afines ha tenido un crecimiento importante, de hecho, existen numerosos esfuerzos de investigación, sin embargo, el cuerpo de conocimiento existente está lejos de estar completo para conducir el desarrollo de sistemas software con dinámica de adaptación.

ORGANIZACIÓN DEL DOCUMENTO

El capítulo 1 tiene por finalidad introducirnos en los elementos que constituyen la metodología elaboración del proyecto, trazando objetivos desagregados en los capítulos posteriores.

El capítulo 2 introduce los conceptos detrás de los sistemas adaptativos complejos que forman parte del marco de referencia del desarrollo.

El capítulo 3 amplia el objeto de estudio mediante la descripción papel del diseño de software y los diferentes conceptos que giran en torno a él.

El capítulo 4 contiene una descripción del proceso de adaptación y su relación con el diseño de software.

El capítulo 5 describe los elementos de la propuesta de marco de trabajo para aplicar la adaptación en la etapa de diseño de un software.

El capítulo 6 muestra la aplicación de la propuesta a un caso de estudio describiendo como introducir la adaptación en el mismo.

Finalmente, el capítulo 7 evalúa sobre el caso de estudio la aplicación de los elementos descritos en el marco de trabajo propuesto permitiendo de esta manera sacar conclusiones al respecto.

1. METODOLOGÍA DE INVESTIGACIÓN

1.1 PROPÓSITO DE INVESTIGACIÓN

Proporcionar un prototipo de marco de trabajo para la aplicación de la teoría de sistemas adaptativos complejos en la etapa de diseño de un proceso de desarrollo de software.

1.1.1 Objetivos específicos

- Caracterizar la dinámica de la adaptación en los sistemas complejos
- Caracterizar el rol central del diseño, en general, y de la arquitectura de software, en particular, para el proceso de adaptación de un sistema software.
- Definir un conjunto de conceptos, prácticas y criterios como referencia, para enfrentar y resolver problemas de índole similar.
- Presentar técnicas para soportar la adaptación basada en una perspectiva centrada en el diseño.
- Realizar una prueba piloto sobre un caso de estudio para verificar la solución y valorar la tesis.

1.2 SITUACIONES DEL PROBLEMA

En el desarrollo de software la etapa de diseño se enfoca principalmente en la toma de decisiones, busca dar respuesta a las preguntas del tipo: *cómo, qué, cuándo y dónde*. Por ejemplo, ¿cómo organizar los ficheros del proyecto?, ¿qué tipo de estructura de datos es más adecuada para la solución de un problema?, ¿dónde debe ubicarse la función $y(x)$?, etc. Conceptualmente, las

herramientas del diseño son[2]: la separación de preocupaciones y responsabilidades, la abstracción, el modularidad y otro “conjunto de principios de ingeniería”, tales como, anticipación del cambio y diseño para generalidad. Por sí mismas, estas herramientas proveen una guía al diseñador, sin embargo, es la experiencia la que permite guiar el uso de dichas herramientas de forma efectiva y oportuna.

Ahora bien, el incremento en los niveles de heterogeneidad de los componentes de software, los frecuentes cambios en el contexto, en las metas y los requerimientos, sujeto a las actuales necesidades de seguridad y rendimiento, incrementa la incertidumbre sobre el diseño final, impactando de manera negativa la puesta a punto de la solución[3] .

1.3 JUSTIFICACIÓN

Desde una arista tecnológica, es necesario abordar el problema del cambio en el software con herramientas teóricas bien fundamentadas como alternativa a las técnicas tradicionalmente empleadas, permitiendo en primera medida integrarse de forma coherente dentro del proceso de desarrollo, y por otro lado validar su utilización práctica dentro del ámbito de aplicación elegido.

Desde una arista académica, la función de la ingeniería de sistemas es guiar la ingeniería de los sistemas complejos, esto en relación con los objetivos trazados por la maestría, siempre buscando la profundización teórica en temas afines a la materia, así, de esta manera el análisis y la evaluación del diseño de software bajo el enfoque de la teoría de sistemas adaptativos complejos

[2] BUDGEN, David. Software design. Pearson education Limited. Segunda edición, 2003.

[3] MAZEIAR, Salehi and TAHVILDARI, Ladan. Self-Adaptive Software. Landscape and research challenges.2009.

ayudará a construir espacios de discusión sobre la manera de hacer más sostenible un sistema en su entorno cambiante.

1.4 PREGUNTAS DE INVESTIGACIÓN

¿Es posible adaptar el diseño de un sistema software distribuido, sin perturbar su funcionalidad, mediante la aplicación de un conjunto de reglas, principios y conocimientos basadas en el estudio de los sistemas adaptativos complejos?

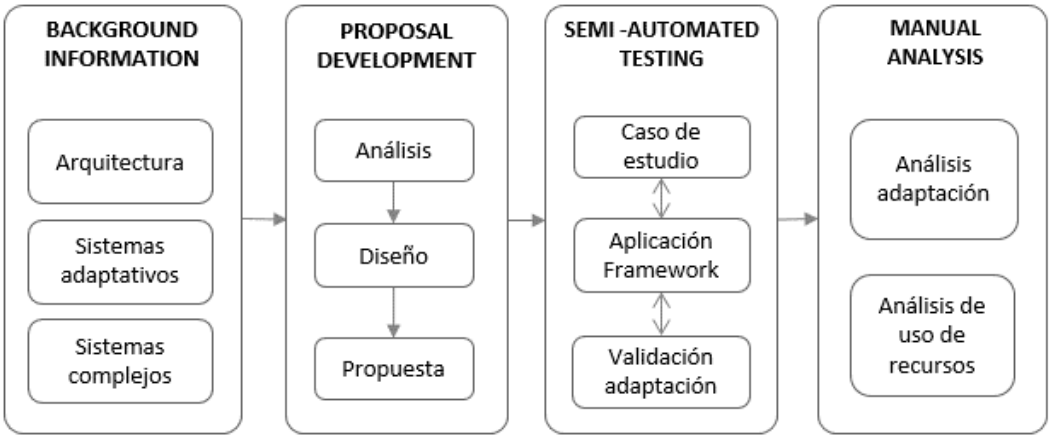
1.4.1 Hipótesis secundarias

- En un diseño adaptable es posible efectuar cambios conceptualizados en términos de entidades que asumen roles específicos en el proceso.
- Las condiciones iniciales del sistema sesgan y limitan los mecanismos de ejecución de la adaptación y es uno de los factores más influyentes al dirigir el sistema hacia un objetivo.
- Se pueden realizar adaptaciones en tiempo de ejecución sin impactar negativamente el rendimiento del sistema.

1.5 ABORDAJE DE LA INVESTIGACIÓN

La metodología general que oriento el proceso de construcción y validación del marco de trabajo se presenta en la Figura 2, está principalmente constituida por cuatro grandes etapas: revisión de antecedentes, desarrollo de la propuesta, pruebas semi-automatizadas y análisis manual de los resultados.

Figura 2. Metodología general del proceso



Fuente. Elaboración personal

Partiendo del tema se construirá un marco teórico de las variables del proyecto, esto incluye, el estudio de los sistemas adaptativos complejos y el diseño de software para una arquitectura distribuida[4], donde se establecerán las posturas teóricas que servirán de referencia conceptual y explicativa en el desarrollo del proyecto, es apropiado utilizar una inmersión bibliográfica extensa y profunda para cumplir con este objetivo.

Dada la necesidad de monitorear el entorno de ejecución, se analizarán los marcos de trabajo desarrollados para aplicaciones *sensibles al contexto*. Con una idea de cómo funcionan, se espera tener un diseño capaz de obtener información del entorno de ejecución, e inyectarla al marco de trabajo que gobierna la adaptación[5].

[4] En el libro "Design Illuminated" se presenta la siguiente definición: "Un sistema distribuido es una colección de dispositivos de almacenamiento y computación conectados a través de una red de comunicación. Los sistemas distribuidos pueden ser modelados por la arquitectura Cliente/Servidor, y esta forma la base para arquitecturas multicapa".

[5] Para mayor información de la acepción particular de este término se puede referir a: Unifying Runtime Adaptation and Design Evolution. Peking University, Beijing, 2009. Donde se construye una conexión causal entre el modelo de diseño abstracto y el sistema en ejecución, donde un cambio en el modelo conceptual desemboca una sincronización con el sistema en ejecución.

Dentro de las características deseables en una arquitectura adaptable está el uso de *políticas*. Las políticas son la representación de comportamientos deseables o restricciones sobre dichos comportamientos que permiten traducir directivas de alto nivel en acciones específicas conducentes a estados deseados, es por lo tanto requerido identificar estrategias que puedan orientar y facilitar este trabajo[6].

Se analizan las estructuras estáticas y dinámicas en la arquitectura distribuida, especificando la organización de los componentes y conectores que la constituyen, su flexibilidad para modificación en tiempo de ejecución. Además, se considera la parametrización de componentes de manera tal que, desde la perspectiva de elementos autonómicos asociados a recursos de computación, tanto la creación de elementos constitutivos, como sus enlaces, puedan ser generados dinámicamente en función de tales parámetros[7].

Una vez realizados los análisis citados, se sintetiza el estudio con la postulación de patrones comunes que exhibieron, luego se propone una caracterización que conlleve a la contextualizan y propuesta de estrategias, técnicas, prácticas y reglas basadas en el marco teórico, para el diseño de software de una arquitectura distribuida. Se espera que el producto de esta síntesis sea la aproximación de un marco de trabajo, disponible para validar su utilidad en una prueba funcional.

Con este esquema metodológico, el proyecto de investigación se divide en cuatro partes:

---

[6] WHITE, Steve. R. An architectural approach to autonomic computing. Se describe un conjunto de ideas

[7] MAGEE, Jeff y KRAMER Jeff. Dynamic Structure in Software Architectures.

1. Dentro del marco teórico se establecen las referencias a los ejes temáticos que darán los antecedentes del estudio: los sistemas adaptativos, los sistemas complejos y la disciplina del diseño de software.
2. En la elaboración de la propuesta de técnicas, prácticas y reglas de aplicación de la teoría de sistemas adaptativos complejos, se construirá la aproximación de marco de trabajo a ser utilizado en la etapa de diseño del software.
3. En la tercera parte, relacionada con la experimentación, en aras de validar y verificar la efectividad del marco de trabajo propuesto se implementará un caso de estudio.
4. Se lleva a cabo una última etapa, donde se exponen los resultados y conclusiones de la aplicación del marco de trabajo, luego de un análisis manual del estado de los recursos y de la adaptación conseguida o no en el caso de prueba.

Esta validación final, se lleva a cabo con una estrategia de evaluación por perspectivas[8] para rendimiento y adaptación que darán la información suficiente para medir el impacto real del marco de trabajo. Con lo mencionado hasta el momento y dada la naturaleza del proyecto, donde la adaptabilidad sugiere modificar para mejorar las condiciones actuales del sistema, el enfoque del estudio será de tipo cuantitativo.

1.6 ALCANCE

La solución propuesta es la aplicación de teoría de sistemas adaptativos complejos al diseño de un marco de trabajo para el desarrollo de sistemas software adaptables. Esto puede llegar a implicar:

- La caracterización de las propiedades de un sistema complejo adaptativo.

[8] Una perspectiva de la arquitectura es una colección de actividades, tácticas y líneas guía que son usadas para asegurar que un sistema exhibe un conjunto relacionado de propiedades de calidad que requieren su consideración a través de un número de vistas de la arquitectura.

- El estudio del diseño de software mediante la revisión bibliográfica de las principales técnicas de diseño existentes y su afinidad con la adaptabilidad.
- La definición clara del ciclo de adaptación, definición de etapas y sus interfaces, buscando la separación de responsabilidades y la futura integración en un sistema software.
- Caracterización de patrones de diseño adaptables y propuesta integradora de dichos patrones.
- La propuesta de un marco de trabajo para el diseño de un sistema software que exhiba características de adaptación.

1.7 EVALUANDO EL MARCO DE TRABAJO PROPUESTO

1.7.1 Variables

- *Resolución*: nivel de detalle donde se detectará el cambio (alto, medio, bajo). Este valor que depende de la *granularidad del elemento mapeado*[9].
- *Tipo de cambio*: es una explicación de en qué consiste el cambio, los valores que esta variable puede tomar se dan en función de las capacidades de cambio del sistema objeto de estudio.
- *Frecuencia de muestreo*: número de muestras de datos de las variables elegidas para monitoreo, tomadas en una fracción de tiempo. Hace referencia a cada cuanto el sistema nos arroja información sobre su estado.
- *Latencia*: tiempo tomado por el sistema para realizar la adaptación una vez se detecta su necesidad.
- *Cantidad de escenarios*: número de situaciones reconocibles como desencadenadores de adaptación.

[9] Nota aclaratoria. Con granularidad se hace referencia al tamaño del elemento. Por poner un ejemplo, un objeto con información de una persona y métodos para manipular esa información tiene una resolución mayor que una estructura de objetos como una lista.

- *Número de configuraciones*: número de posibles grafos entre elementos de la arquitectura.
- *Impacto de la adaptación*: son un conjunto de medidas del desempeño de la aplicación para medir el grado de afectación positiva/negativa en la operación del sistema.
- *Tipos de entidades de diseño adaptables*: son aquellas entidades cuya morfología habilita características de cambio y adaptación.
- *Nivel de adaptación*: grado en que las características de entidades y ellas mismas pueden llegar a ser modificadas para adaptación.

Una vez se llegue al capítulo de evaluación se procederá a decidir cuáles de las variables anteriores se aplicarán a la validación del marco de trabajo propuesto.

1.7.2 Instrumento

Se emplea un artefacto similar al de la Figura 3 para evaluar la adaptación.

Figura 3. Instrumento de evaluación

TABLA DE ADAPTACIÓN			
Nombre		**Código**	
ESCENARIO			
Descripción de las condiciones externas.			
Descripción de las condiciones internas.			
Estado elementos (antes de la adaptación)			
Estado elementos (después de la adaptación) Solo para registros que utilizan el marco de trabajo			
IMPACTO			
Variable	**Estado inicial**	**Estado final**	

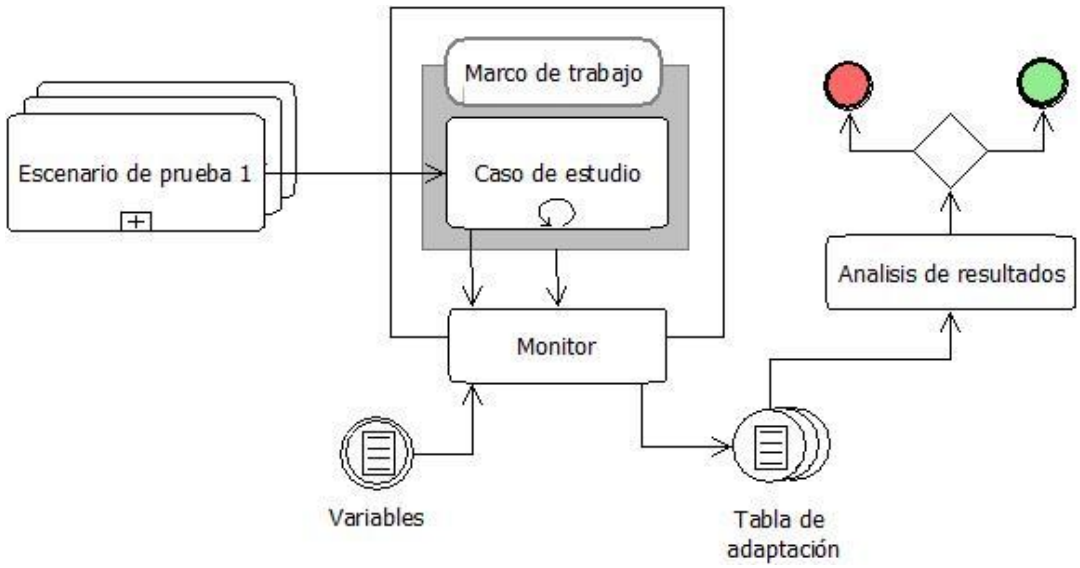
Fuente. Elaboración personal

1.7.3 Análisis de resultados

En términos generales, luego de aplicar los elementos desarrollados como parte del marco de trabajo propuesto al caso de estudio de prueba, para llevar a cabo la etapa de análisis se plantea desarrollar escenarios de prueba para

desencadenar las situaciones bajo las cuales se pueda o requiera realizar una adaptación, la imagen de la Figura 4. aclara el proceso.

Figura 4. Proceso de validación del marco de trabajo



Fuente. Elaboración personal

En primera instancia se crea una colección de *escenarios de prueba* prediseñados para generar situaciones que motiven una adaptación por parte del *marco de trabajo*, luego de ejecutar cada escenario sobre el *caso de estudio* un elemento monitor que puede reducirse a un simple registro por salida estándar para el sistema extrae información de las variables descritas permitiendo de esta manera completar la tabla de adaptación de la sección anterior. El análisis de resultados considera si la situación de cambio es reconocida por el marco de trabajo y ésta desencadena una adaptación y evalúa el impacto de esta. La idea sería realizar una comparación directa entre los estados de configuración a priori y a posteriori la adaptación igual que a las variables de medición del impacto, así determinar la efectividad del marco de trabajo, en el capítulo 7 de este documento se retoma este proceso y se aclarará cualquier inquietud pendiente respecto a sus actividades.

2. SISTEMAS ADAPTATIVOS COMPLEJOS[10]

Los sistemas adaptativos complejos son un tipo de sistema donde la complejidad se asocia con la heterogeneidad y diversidad de los elementos que lo constituyen y que además se encuentran interconectados realizando continuos intercambios de recursos entre sí. La característica de adaptabilidad se presenta cuando dicho sistema exhibe capacidades para cambiar su estructura y comportamiento a partir de la experiencia y de las interacciones con su entorno.

Según GELL-MANN (1994) el concepto de complejidad se refiere al volumen de la de las interacciones entre los elementos constituyentes de un sistema, y tiene por finalidad describir propiedades de este[11].

Cuando se define un *tipo de complejidad* es necesario acotar el nivel de detalle de las interacciones que se analizan, esta actividad es comúnmente conocida como *definición de la resolución*, y puede implementarse como un parámetro utilizado por los elementos de *monitoreo*, una buena definición de este parámetro permite ignorar los detalles más finos, que no aportan ningún valor al proceso de adaptación. Por lo regular se puede identificar varios tipos de complejidad, pero para los intereses de este apartado únicamente se explican dos de ellos, la *complejidad bruta* y la *complejidad efectiva*.

La **complejidad bruta** asociada a un sistema se define como *la longitud de su descripción*. Esta descripción explica en un lenguaje preestablecido todas las interacciones entre los elementos del sistema, esto implica que en alguna

[10] Este apartado resume las ideas fundamentales de los sistemas adaptativos complejos, consignadas en el libro El Quark y el Jaguar escrito por el reconocido físico del Santa Fe Institute, Murray Gell-Mann (1929 - 2007).

[11] Nota aclaratoria. La complejidad en este contexto no se debe vincular con la acepción computacional, donde el interés prima en saber cuánto tardaría, un ordenador en resolver un problema.

medida depende del vocabulario disponible para efectuar la descripción, en otras palabras, si se va a medir la complejidad bruta del sistema, primero se hace necesario identificar el conjunto de variables que represente la interacción entre sus elementos, luego definir el dominio y rango de cada una de ellas, finalmente definir un nivel de resolución adecuado que facilite su posterior análisis.

Si la complejidad bruta se define en términos de la longitud de una descripción, entonces según GELL-MANN (1994) no es una propiedad intrínseca de la cosa descrita, sino que obedece a las capacidades y limitaciones del descriptor, es decir, el *agente* que realiza la descripción. Así, la longitud de la descripción dependerá del lenguaje empleado tanto como del conocimiento o concepción del mundo que compartan los interlocutores, en este caso los *agentes* encargados manejar las aserciones del sistema. Resumiendo, la c*omplejidad bruta* se define como, la *longitud del mensaje más corto que describe un sistema, con una resolución dada, dirigido a un interlocutor lejano y haciendo uso de un lenguaje y un conocimiento del mundo que tanto emisor como receptor comparten (y saben que comparten) con anterioridad*.

La **complejidad efectiva** de un sistema se define como *la longitud del esquema utilizado para definir sus regularidades*. El término *esquem*a se emplea para referirse a una estructura conceptual de la que el ser humano hace uso para comprender un conjunto de datos, en otras palabras, es un modelo que puede ser gráfico o formal, como un sistema de ecuaciones, un diagrama de frecuencias, etc. Cuando se habla de *complejidad efectiva interna* se hace una extensión del concepto para referirse a un *esquema* que gobierna de algún modo el sistema, en vez de limitarse a ser solo un recurso de un observador externo para describir el sistema.

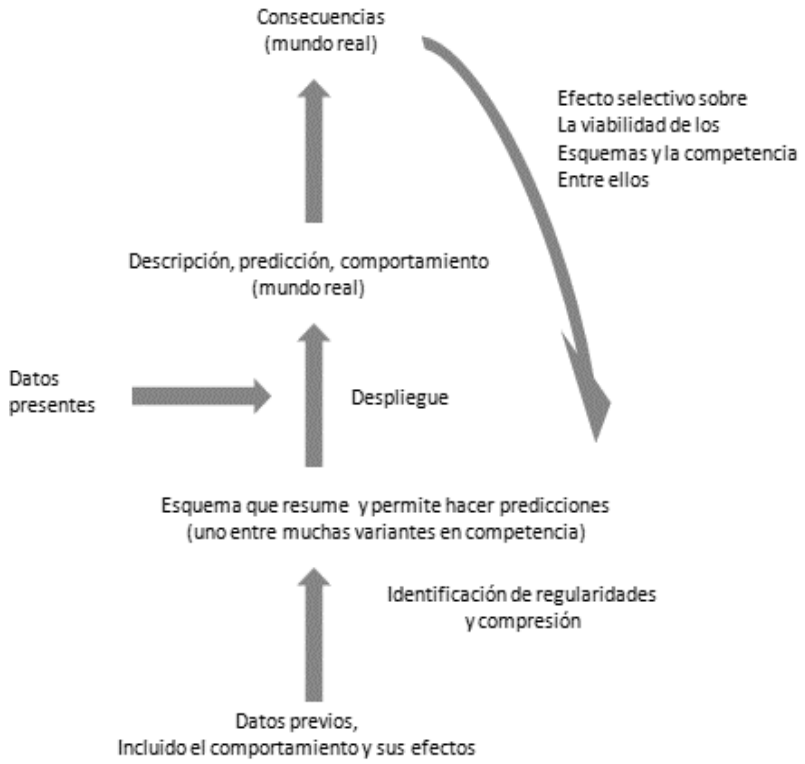
La complejidad efectiva es útil cuando una entidad externa recoge observaciones del sistema, y es competente para identificar y comprimir en un

modelo *regularidades*, y descartar lo que es meramente incidental. Si no es de esta manera, la complejidad efectiva del sistema observado tiene que ver más con las limitaciones del observador que con las propiedades del sistema observado.

Lo anterior plantea el siguiente cuestionamiento, *¿está realmente bien definido el problema de identificar regularidades a partir de una sucesión de datos?* La respuesta es que entre mayor sea el número de observaciones, será más fácil encontrar *regularidades*. Sin embargo, esto plantea otra cuestión, *¿es la información obtenida del sistema observado, suficiente para construir un esquema desde cero?*, la respuesta es no, la única fuente plausible de dicha información es una tendencia innata, es decir, un primer esquema parcialmente arraigado con anterioridad en el sistema observador, dicho esquema inicial es producto de un conocimiento declarativo previo sobre el sistema observado.

La imagen de la Figura 5. puede dar claridad sobre el funcionamiento los sistemas adaptativos complejos.

Figura 5. Sistema adaptativo complejo, ciclo de adaptación



Fuente. El quark y el jaguar.

En general la ventaja de considerar una *sucesión de datos* indefinidamente larga es que se incrementa enormemente la posibilidad de discriminar entre modelos distintos, cada modelo corresponde a una clase particular de regularidades, identificadas previamente en el cuerpo de experiencia de un esquema.

Una *clase de regularidades* corresponde a un conjunto de modelos que describen o generan la sucesión de datos. Cuando un sistema adaptativo complejo recibe una sucesión de datos arbitrariamente larga, puede dedicarse a buscar sistemáticamente regularidades de una clase dada, cualquier regularidad identificada puede incorporarse en un sistema que describa la sucesión de datos o un sistema que pueda generar dicha sucesión.

El estudio de cualquier *sistema complejo adaptativo* se concentra en la *información*, esta llega al sistema en forma de flujo de datos. La gráfica describe la manera en que el sistema percibe regularidades que extrae del flujo de datos separándolas de lo que es meramente incidental o arbitrario y condensando estas en un esquema sujeto a variación. A partir de allí cada esquema resultante se combina con información adicional, proveniente de aquella información dejada de lado en la abstracción de regularidades, para generar un resultado aplicable al mundo real: descripción del sistema observado, la predicción de algún suceso o la prescripción del comportamiento del propio sistema.

Finalmente, se observa los efectos de dicho resultado en el mundo real; tales efectos son retroactivos, ejerciendo "*presiones selectivas*" sobre los esquemas en competencia y permitirán que algunos se mantengan y otros sean descartados.

Este primer acercamiento a la definición de sistema adaptativo complejo introduce un conjunto de ideas básicas para el diseño del marco de trabajo y que se resume brevemente a continuación:

- La complejidad se asocia con las interacciones entre elementos de un sistema.
- Entre mayor sea la cantidad de tales interacciones hay mayor posibilidad de identificar "regularidades", en adelante se llamará a estos: patrones.
- El sistema adaptativo complejo puede ubicarse como un subsistema del sistema objeto de estudio, o puede ser un sistema independiente que hace las veces de un observador.
- Basado en las características propias de un sistema de software, que está constituido por una diversidad de entidades que establecen relaciones e interacciones continuas con otras dentro del mismo

sistema, se quiere evaluar la posibilidad de monitorear esta complejidad subyacente y extraer de ella información a partir de la cual se pueda hacer una descripción del sistema observado, la predicción de algún suceso o la prescripción del comportamiento del propio sistema.

La siguiente sección introduce otras ideas desde un área de estudio emergente conocida como la *computación autonómica*, ésta aporta una aproximación al diseño de sistemas adaptativos complejos, dicha aproximación se basa en el estudio de sistemas constituidos por agentes.

2.1 LA COMPUTACIÓN AUTONÓMICA[12]

En octubre de 2001, la IBM publicó el *Manifiesto Horn*[13] en este documento la compañía señala cuáles serían los principales obstáculos en el progreso de la industria de tecnologías de la información (TI, en adelante) durante las siguientes décadas. La compañía citó como uno de los obstáculos del devenir de aplicaciones y entornos de millones de líneas de código que requerirán profesionales con habilidades especializadas para instalación, configuración, afinamiento y mantenimiento. Este manifiesto apunta que la mayor dificultad en la administración de los sistemas de cómputo va más allá de la administración individual de los entornos de software; y debe considerar la *heterogeneidad* de los sistemas y la tendencia actual de las compañías a encaminarse más allá de los límites tradicionales, introduciendo nuevos niveles de complejidad en la misma.

Como se mencionó en la anterior sección, la complejidad de un sistema se define en función de volúmenes altos de interacción entre sus elementos constituyentes, y en la actualidad el manejo de esta complejidad supera el límite de la capacidad humana. Los sistemas de cómputo actuales han

[12] Las notas consignadas en este apartado obedecen a la traducción del artículo. IBM Research. An architectural blueprint for autonomic computing. June 2005
[13] Paul Horn: Autonomic Computing, IBM's Perspective on the State of Information Technology

incrementado el número de interacciones entre sus elementos, y aun cuando un especialista en *arquitectura de sistemas* puede diseñarle, si el sistema llega a ser más diverso y con volúmenes grandes de interconexiones e interacciones, es menos probable, que dicho especialista pueda anticipar y diseñar todos los escenarios de interacción entre sus componentes, dejando que muchos temas sean abordados cuando el sistema se encuentra en ejecución.

Los sistemas son cada vez más masivos, y debido al rápido flujo de cambios y demandas en conflicto, no hay manera de tener herramientas decisivas a tiempo. Como solución a la complejidad en aumento de los sistemas surge la idea de dotar estos con la capacidad de *administrarse a sí mismos,* sin intervención humana directa[14]. De lo anterior surge la idea de la *computación autonómica* que hace referencia *al campo de estudio y desarrollo de sistemas que pueden administrarse a sí mismos en función de objetivos de alto nivel*[15].

Como se verá más adelante, el modelo subyacente que gobierna a la *computación autonómica* es similar al presentado en la sección de *sistemas adaptativos complejos*, la diferencia radica en la granularidad de los elementos que realizan la adaptación y esto repercute en la adición de un nuevo nivel de interacciones que gobierna el sistema, y que en función del volumen de estas puede exhibir o no características de complejidad.

Cuando el vicepresidente de investigación de IBM, Paul Horn introdujo dicha idea, deliberadamente eligió un término con connotación biológica, refiriéndose al sistema nervioso autónomo humano. El sistema autónomo, es la parte del sistema nervioso que controla las acciones involuntarias tan importantes como el ritmo cardíaco, la temperatura del cuerpo, la presión de la sangre, la dilatación de las pupilas; y libera al cerebro *consciente* de la carga de manejar éstas y otras "*acciones de bajo nivel"*, pero que son aun vitales para el

[14] HUEBSCHER, Markus C y MCCANN, Julie A. A survey of Autonomic Computing — degrees, models and applications. Imperial College London. ACM Computing Surveys (CSUR), 2008

[15] IBM Research. An architectural blueprint for autonomic computing. June 2005

funcionamiento del cuerpo humano[16]. El término también hace referencia a una vasta jerarquía de sistemas naturales autogobernados que van desde nano-moléculas en las células hasta entidades en el mundo socioeconómico.

Una aproximación desde la ingeniería de sistemas presenta un sistema autónomo como aquel que mantiene y ajusta su operación de cara a cambios en sus componentes, cargas de trabajo, demanda, condiciones externas, y fallas en el hardware o en el software. El paradigma de la computación autonómica debe contener mecanismos donde los cambios en sus variables esenciales puedan desencadenar cambios en el comportamiento del sistema, de modo tal, que éste puede ser llevado a un estado de equilibrio con respecto al entorno, dicho estado es una condición necesaria para la supervivencia del organismo. Para el caso de un sistema de cómputo la "*supervivencia*" puede relacionarse con habilidades de protección a sí mismo, recuperación por fallas, reconfiguración por cambios en el entorno y mantenimiento en operación en niveles aceptables de rendimiento, por lo anterior este tipo de sistemas exhibe lo que se conoce como autoadministración.

Es posible pensar en comportamientos de alta especialización en un sistema autónomo, como permitir que cuando se introduzca un nuevo componente en el sistema, el resto de éste se adapte a su presencia de forma natural igual que una célula al cuerpo, o como una persona a una población; con el monitoreo y experimentación de su estado interno, con el refinamiento de parámetros, el sistema exhibe aprendizaje tomando decisiones inteligentes sobre cuáles funciones conservar o cuales eliminar. El sistema podría incluso diagnosticar problemas y con su información de configuración emparejar el diagnóstico contra parches que solucionen el problema e instalarlas en tiempo de ejecución de forma automática.

IBM ha definido cinco niveles de madurez para caracterizar la inyección gradual de autonomía en sistemas software.

[16] "Sistema Nervioso Autónomo". *Es.wikipedia.org*. N.p., 2016. Web. Apr. 2016.

- Nivel básico: análisis y diagnóstico manual, y solución de problemas.
- Nivel administrado: herramientas centralizadas acciones manuales.
- Nivel predictivo: correlación multi-recurso y guía de soluciones.
- Nivel autónomo: dinámico, basado en reglas de negocio, administrado.
- Nivel adaptado: monitoreo de sistema, correlación y toma de decisión.

Todo lo anterior presenta la *computación autonómica* como una buena aproximación de diseño de sistemas que exhiben complejidad, al considerar elementos autónomos sensibles al contexto con posibilidad de decisión y capacidad de interacción con otros; si estos elementos tienen objetivos claros pueden conducir al sistema observado hacia estados deseables de operación. En las siguientes secciones se describe con mayor detalle elementos importantes de este tipo de sistemas para la construcción de un marco de trabajo en este sentido.

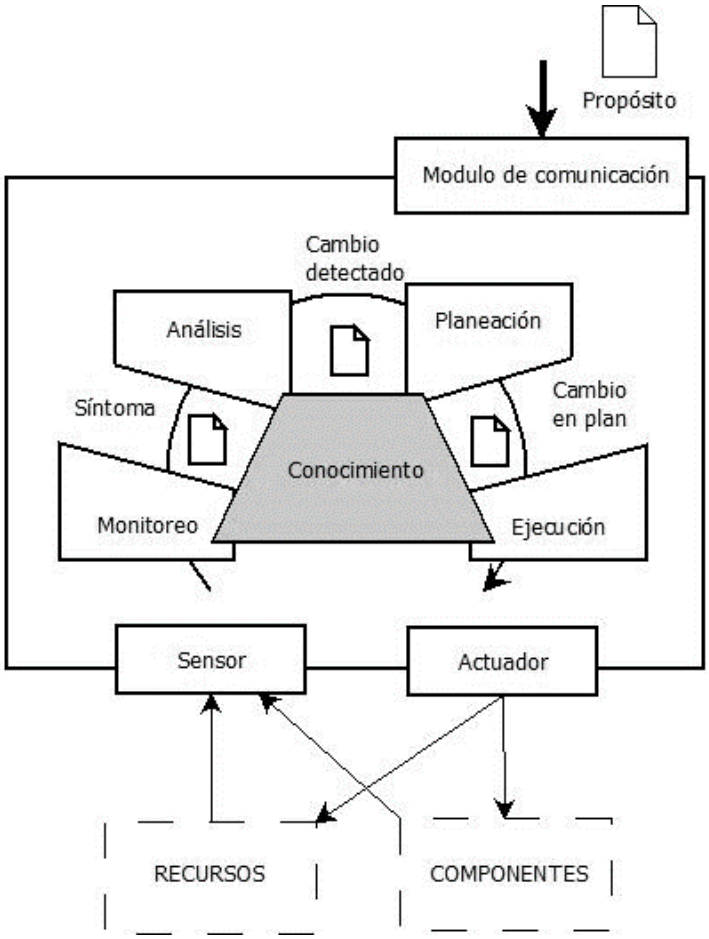
2.2 CICLO DE CONTROL[17]

En el corazón de todo sistema autonómico está el ciclo de control, el cual es una combinación de componentes que actúan juntos para mantener los valores de los atributos del actual sistema cerca de un conjunto de *valores de referencia preestablecidos*. El ciclo de control de un elemento en la computación autonómica es, en esencia, una aproximación al diseño del ciclo de adaptación descrito para un sistema adaptativo complejo Figura 5. A continuación, se brinda mayor detalle respecto a este diseño y al final de la sección se hace algunas precisiones contra el modelo especificado en la Figura 6.

[17] HUEBSCHER, Markus y McCANN, Julie A. Evaluation issues in autonomic computing. Department of computing, Imperial College London. 2003.

Un sistema de *ciclo de control cerrado*[18] es aquel en el cual las salidas tienen un efecto en el comportamiento de este, de esta forma el sistema busca a *mantener un valor deseado* en la salida. Un sistema autónomo puede emplear uno o más ciclos de control cerrado. En concreto, un sistema autoadministrado monitorea algunos recursos y de manera autónoma intenta mantener ciertos parámetros de operación en un rango deseado.

Figura 6. Modelo de elemento autonómico



Fuente. An architectural blueprint for autonomic computing.

[18] Nota aclaratoria. Un sistema provisto de los tres elementos de control (objetivo, mecanismo de control y ciclo de retroalimentación) es un sistema de ciclo cerrado. Extraído de "SISTEMA DE CICLO ABIERTO Y CICLO CERRADO - Enciclopedia De Tareas". *Enciclopediadetareas.net*. N.p., 2016. Web. May 2016.

La imagen en la Figura 6. presenta la aproximación a un diseño de alto nivel de la estructura de un elemento autonómico.

Un bloque fundamental de construcción de un sistema autonómico es la *capacidad sensorial*. La capacidad sensorial permite a la entidad autónoma observar su *contexto operacional* y el *comportamiento y/o estado* de sus componentes, es decir, a través de éste el elemento autónomo realiza observaciones con cierta frecuencia de su entorno (recursos) y de su estado interno (estado de sus componentes) convirtiendo estas en un flujo de datos "permanente", susceptible de *análisis*. Los datos "*en crudo*" son registrados, bajo ciertos parámetros configurables de muestreo, y son resumidos en *esquemas descriptivos*[19] por el componente de monitoreo.

El componente de *monitoreo* puede encontrar a partir de estos esquemas, desviaciones respecto de registros históricos, sugiriendo comportamientos anómalos o fuera de la "*cotidianidad*" del componente analizado. En otras palabras, el *monitor* observa el flujo continuo de datos obtenido por los sensores, filtra los datos recolectados, y extrae información en forma de "*síntomas*" que luego puede almacenar y compartir.

Los "*síntomas*" son analizados y correlacionados con el objetivo de identificar situaciones, conductas o patrones en que el sistema o alguno de sus componentes sea conducido a los límites de su propósito de operación y se pueda emitir advertencias de cambio en el *comportamiento o estado* de un recurso o componente del sistema. En otras palabras, el *motor de análisis* compara los datos colectados contra valores deseados o esperados y emite eventos de detección de cambios.

El *motor de planificación* diseña una estrategia para corregir las desviaciones identificadas, su función es garantizar la continuidad de la operación del

[19] Nota aclaratoria. Se utiliza este término extraído de la teoría de sistemas adaptativos complejos con el ánimo de trazar relaciones entre ambas áreas de estudio, que posteriormente sirvan de sustento teórico a la propuesta de marco de trabajo.

sistema durante el proceso de adaptación a los factores endógenos y exógenos detectados. Apoyado por el conocimiento compartido, puede crear un plan de cambio, es decir, una secuencia de instrucciones bien definida.

Finalmente, la *máquina de ejecución* lleva a cabo las acciones descritas en el plan, estas pueden variar desde el simple ajuste de parámetros de los elementos administrados, hasta cambios más severos como la sustitución de un componente por otro. Gracias a la planificación y mediante el uso de efectores que operan directamente sobre el recurso, se puede alcanzar una suerte de *homeóstasis*[20] para el sistema. En cierta medida cada componente debe ser responsable por proporcionar los mecanismos efectores de la adaptación para servir al propósito del sistema que lo hospeda. Esto quiere decir, el componente administrado debe hacer pública su interfaz de adaptación.

Una aplicación de software eficaz emerge de la sinergia entre el conjunto de sus componentes trabajando como un todo y dirigidos hacia una meta, por su parte una aplicación de software eficiente emerge durante la operación de dicho sistema bajo las restricciones impuestas por su entorno.

Un elemento autonómico administra su estado interno y sus interacciones con el entorno. Su comportamiento y relaciones con otros elementos pueden organizarse en diversas topologías, y son conducidos por metas y políticas. Cabe resaltar que este modelo tiene una connotación teleológica[21] en su diseño, lo cual se evidencia en la suma de la intención derivada del propósito (misión, políticas, instinto de supervivencia) y el conocimiento creado a lo largo del ciclo de vida del elemento.

Hay tres maneras básicas de diseñar un sistema autonómico:

[20] “La homeostasis es la tendencia de los organismos vivos y otros sistemas a adaptarse a las nuevas condiciones y a mantener el equilibrio a pesar de los cambios.” Extraído de: "Homeostasis - Ecured". *Ecured.cu*. N.p., 2016. Web.

[21] Nota aclaratoria. Este concepto expresa un modo de explicación basado en causas finales. Aristóteles y los Escolásticos son considerados como teleológicos en oposición a las causalistas o mecanicistas.

1. Diseñar el sistema para soportar un “*espacio de posibles comportamientos*” es una estrategia orientada al diseño por escenarios muy común en sistemas de información donde el analista considera flujos de interacciones entre el usuario y el sistema y los presenta en lo que entre la comunidad de desarrolladores se conoce como caso de uso. Este modelo es en pocas palabras determinista y la adaptabilidad del sistema está condicionada a la calidad de los escenarios.
2. Dotar al sistema con capacidades de planeación y habilidades sociales, de esta forma el sistema puede delegar tareas a componentes de software externo (agentes) aumentando sus propias capacidades. Este tipo de diseño faculta al sistema con capacidades de autoorganización y puede compartir información con el medio externo.
3. Diseñar una aproximación evolutiva basada en los modelos biológicos. Esta aproximación es conocida como el paradigma de computación orientada a la autonomía, este usa sistemas artificiales imitando comportamientos colectivos de sociedades de animales para resolver problemas de difícil computación.

Las anteriores aproximaciones pueden hacer uso de mecanismos como patrones de diseño, patrones de arquitectura, y estilos arquitectónicos que han sido bien estudiados y son conocidos en la comunidad de desarrollo de software, estos pueden llegar a simplificar el diseño del sistema autonómico, el próximo capítulo dará el detalle suficiente sobre este tema.

Una de las propiedades más importantes de un sistema autonómico es la separación espacial y temporal, esto se debe entender como la restricción de separar el sistema controlador de los elementos controlados, la principal razón es permitir la evolución del controlador y tener mediciones apropiadas del recurso controlado.

2.3 ATRIBUTOS ADAPTATIVOS[22]

A pesar de que el *propósito de un sistema* es único y varía de un sistema a otro, todos los sistemas autonómicos exhiben un mínimo conjunto de propiedades[23], que se mencionan a continuación:

- Automático: significa que el sistema debe tener la capacidad de controlar sus funciones internas y operaciones.
- Auto-sensibilidad: significa que el sistema tiene "*conocimiento*"[24] de sus componentes, de su estado, funciones e interacciones con el entorno.
- Auto-adaptabilidad: significa que el sistema flexibiliza su estructura o su comportamiento para encontrar formas de mejorar su interacción con sistemas vecinos o con su entorno.

Múltiples trabajos[25] en esta área han extendido este conjunto mencionado de características a:

- Autoconfiguración: significa que el sistema tiene la capacidad para cambiar las políticas de alto nivel y así iniciar acciones de ajuste automáticamente por el resto del sistema.
- Auto-optimización: significa que el sistema y sus componentes (como un todo), continuamente exploran oportunidades para mejorar su rendimiento y su eficiencia para alcanzar los objetivos trazados.
- Auto-sanación: significa que el sistema de manera autónoma detecta, diagnostica y repara los problemas localizados.

---

[22] M. Salehie and L. Tahvildari. Autonomic Computing: Emerging Trends and Open problems. ACM, 2005

[23] LAPOUCHNIAN, Alexei y LIASKOS, Sotirios y MYLOPOULOS, Jhon y YU Yijun. Towards Requirements-Driven Autonomic System Design. Department of Computer Science University of Toronto. 2005.

[24] Nota aclaratoria. En este contexto se está más interesado en interpretar este *conocimiento* como la capacidad de respuesta a ciertos estímulos.

[25] WHITE Steve R., HANSON James E., WHALLEY Ian, CHESS David M., y Jeffrey O.Kephart. An architectural approach to Autonomic computing. IBM Thomas J.Watson Research Center. 2004.

- Autoprotección: significa que el sistema se defiende contra fallas en cascada o ataques de seguridad.

En la actualidad, como resultado de continuos estudios se ha expandido el conjunto de características auto-*, entre las cuales cabe mencionar algunas de indudable importancia:

- Auto-regulación: significa que el sistema en cuestión opera para preservar algunos parámetros en rangos de valores adecuados, un ejemplo serían los parámetros de calidad de servicio.
- Autoaprendizaje: significa que el sistema emplea tecnologías y técnicas para extraer y conservar conocimiento de su experiencia.
- Auto-creación: (auto-ensamblaje, auto-replicación) el sistema es conducido por modelos sociales y ecológicos sin presión explícita desde fuera del sistema. Los miembros del sistema generan complejidad y orden en respuesta creativa a la continua demanda estratégica de cambio.

2.4 POLÍTICAS[26]

Las políticas son un conjunto de consideraciones diseñadas para guiar las decisiones que afectan el comportamiento de tareas sobre recursos administrados. Se distinguen tres tipos de políticas: políticas de acción, políticas de meta, políticas de funciones utilitarias.

Las *políticas de acción* indican que la acciones que debería tomar el sistema cuando se encuentra en un estado dado, estas tienen la forma condicional: ***si*** *(condición)* ***entonces*** *(acción)*, estas políticas se diseñan con el propósito de asegurar que el sistema tenga un comportamiento racional.

Las *políticas de meta* especifican un estado deseado, incluso uno o más criterios que caracterizan a un conjunto de estados deseado.

[26] MÜLLER, Hausi A., et al. *Autonomic computing*. CARNEGIE-MELLON UNIV PITTSBURGH PA SOFTWARE ENGINEERING INST, 2006.

Las *políticas de función de utilidad* se modelan mediante una función objetivo que expresa el valor de un posible estado en función de varias variables, su especificación es más flexible, son la base de la teoría de optimización.

Kephart y Walsh[27] proponen un marco de trabajo unificado para políticas para un sistema de computación autonómica basado en máquinas de estado donde la política directa o indirectamente causa la transición a nuevos estados, tal transición es producto de un conjunto de acciones llevadas a cabo por el sistema.

2.5 TAXONOMÍA DEL CAMBIO[28]

Como ya fue mencionado "*el cambio es el endémico del software"*. Los usuarios cambian sus ideas acerca de lo que desean e incluso lo que requieren de una aplicación. Los diseñadores buscan mejorar sus diseños con respecto al rendimiento, apariencia u otras propiedades. El entorno de una aplicación cambia continuamente, se crean y eliminan procesos, se publican y consumen servicios, se hace uso continuo de recursos de red o de sistema de archivos, etc. Sin importar la razón del cambio, los desarrolladores de software encaran el reto de la continua necesidad de modificar o ajustar una aplicación. Todos estos "*tipos de ajustes*" se pueden agrupar bajo la sombrilla del término adaptación como: *modificación de un sistema software para satisfacer nuevos requerimientos y circunstancias cambiantes*.

Varias son las motivaciones para un cambio, quizá la motivación más común es el *cambio correctivo*, en tal situación la aplicación es adaptada para *satisfacer* un requerimiento existente, en otras palabras, corregir defectos. En

[27] KEPHART, Jeffrey O. & WALSH, William E. "An Artificial Intelligence Perspective on Autonomic Computing Policies," 3-12. Proceedings of the Fifth IEEE International Workshop on Policies for Distributed Systems and Networks (Policy 2004). Yorktown Heights, NY, June 7-9, 2004. Los Alamitos, CA: IEEE Computer Society, 2004.
[28] TAYLOR, Richard N. y MEDVIDOVIC, Nenad y DASHOFY, Eric M. Software Architecture. Foundations, Theory and Practice. Wiley, 2010

este tipo de cambios es esencial entender la causa para determinar cómo corregir el defecto.

Una segunda motivación para un cambio es la *modificación de un requerimiento funcional,* ya sea por la adición de una nueva característica, la modificación de una característica existente, o quizá en algunos casos la remoción de alguna. Esta situación se presenta con frecuencia una vez la aplicación entra en funcionamiento, debido a que el uso de la aplicación motiva a los usuarios a pensar en sus tareas de formas nuevas y diferentes.

Una tercera motivación para el cambio es satisfacer propiedades no-funcionales del sistema, nuevas o modificadas. Una nueva propiedad no-funcional puede inducir cambios profundos en la arquitectura, como un cambio de tipo *mejor organización* – reestructurando una aplicación en anticipación a requerimientos para cambios futuros. Por ejemplo, una nueva plataforma middleware puede ponerse a disposición como un paso proactivo para soportar potenciales modificaciones futuras, esto puede implicar un cambio profundo en la línea base de la arquitectura de software.

Una cuarta motivación para el cambio es la necesidad de satisfacer entornos de operación cambiantes, en esta situación las funciones de la aplicación son alteradas, es decir, la estructura de la aplicación debe adaptarse para la ejecución. Esto deriva en un tipo de adaptación que será discutida en detalle en capítulos posteriores, conocida como adaptación *on-the-fly*[29], o adaptación en tiempo real.

Finalmente, un último tipo de motivación es aquella derivada de las actividades de *observación y análisis*. Considere el proceso de observar las propiedades y comportamientos del sistema y luego compararlas contra el propósito (*metas, objetivos, etc.*) definido para el sistema, y mediante un análisis de las observaciones se revelen discrepancias que puedan sugerir el inicio de actividades de adaptación.

[29] "On The Fly". *Wikipedia*. N.p., 2016. Web. July 2016.

2.6 CONCLUSIONES

Este capítulo determina consideraciones importantes para el desarrollo futuro del trabajo propuesto, los siguientes puntos tratan de resumir estas:

- Al observar con detalle el modelo propuesto para sistemas autonómicos y compararlo con la definición y teoría relacionada al sistema complejo adaptativo las coincidencias en ambos modelos saltan a la vista, ambos se pueden acoger a la definición: "*Sistemas abiertos al entorno, de ciclo cerrado que alteran su estructura o comportamiento de cara a cambios y metas*". Sugiere lo anterior que no son áreas de trabajo ajenas la una a la otra y que tal vez sea útil analizar la posibilidad de considerarlos como un sistema de sistemas, esto introduce una estructura jerárquica con propiedades que vale la pena analizar en profundidad.
- Al considerar el apartado de clasificación del cambio se hace necesario que la propuesta delimite los tipos de cambio que se quieren considerar en el desarrollo del proyecto, dada la existencia de diferentes niveles de abstracción del proceso de desarrollo del software, considérese (análisis, diseño, implementación, pruebas) y la diversidad de artefactos que cada nivel maneja, se propone estudiar en detalle los cambios a nivel de entornos de operación. Esta aproximación constituye el hilo conductor del trabajo, pero no quiere decir que no se considerarán los otros tipos.
- Un esquema definido de políticas es requerido para conducir el sistema hacia mejores estados de ejecución, se deben analizar con detalle los diferentes tipos de políticas y adoptar uno o una combinación de ellos acorde a los requerimientos propios del marco de trabajo propuesto.

3. EL DISEÑO EN EL DESARROLLO DE SOFTWARE

Al hablar de *diseño de software* se hace referencia a aquella etapa del ciclo de vida del desarrollo de software donde los involucrados (diseñadores, programadores) toman decisiones, y donde estas decisiones pretenden dar respuesta a las preguntas de tipo *¿cómo?* Quizá un mejor punto de partida sea considerar las palabras de (Budgen, 2003).

"*El propósito del diseño es simplemente producir una solución a un problema. Los diseñadores necesitan aprender a pensar acerca de un sistema de una forma abstracta - en términos de eventos, entidades, objetos, o cualquier otro mecanismo apropiado - y dejar cuestiones de detalle, para etapas posteriores del desarrollo.*"

El diseño de un sistema puede ser dividido en dos niveles; por un lado, el correspondiente con el conjunto de decisiones de alto nivel, comúnmente agrupadas en lo que se conoce como *arquitectura del software*, este conjunto de decisiones incluye decisiones de diversa índole, como por ejemplo: componentes principales de la aplicación, descripciones detalladas de los conectores que pueden incluir tipos y restricciones de interacción entre los componentes, configuraciones entre componentes y conectores, que pueden ser representadas mediante grafos, además de patrones y estilos arquitectónicos[30].

Un *componente* es una entidad de la arquitectura que encapsula un subconjunto de funcionalidades y/o datos. Un componente puede pertenecer a

[30] Para mayor detalle, referirse a: TAYLOR, Richard N. y MEDVIDOVIC, Nenad y DASHOFY, Eric M. Software Architecture. Foundations, Theory and Practice. Páginas 57 - 81. Wiley, 2010

diferentes ámbitos[31] y organizarse en configuraciones relacionado con otros componentes o conectores, en lo que se conoce como *vistas de la arquitectura*, al tratarlo de esta manera un *arquitecto de software* puede enfocar su atención en un área específica de interés para el desarrollo. Una metodología comúnmente usada para extraer la arquitectura de software de la especificación funcional de sus requerimientos es el *Modelo 4+1 de Kruchten*[32], en tal modelo la idea principal es partir de un subconjunto de *casos de uso*[33] priorizados, que sea representativo de la totalidad de las formas de uso del sistema por parte de todos usuarios (actores que pueden representar personas u otros sistemas), a partir de allí el arquitecto elabora el siguiente conjunto de vistas: *vista lógica, vista de despliegue, vista de procesos y vista física* del sistema que se está estudiando. En conclusión, la arquitectura emerge como resultado de un proceso de análisis y "*proyección*" de las principales funcionalidades del sistema sobre cada una de las vistas (dimensiones) consideradas, la imagen en la Figura 7 resume estas ideas.

[31] Nota aclaratoria. Puede hacer alusión a elementos lógicos o físicos. Algunos ejemplos son, procesos en ejecución en sistema operativo, nodos de procesamiento en una red de computadoras, etc.
[32] KRUCHTEN, Philippe. "Planos Arquitectónicos: El Modelo de 4 + 1 Vistas de la Arquitectura del Software." *IEEE Software* 12.6 (1995): 42-50.
[33] Definición. Los casos de uso son considerados formas de usar el sistema por parte de agentes externos a éste conocidos como actores. La especificación de un caso de uso permite separar las responsabilidades del actor de las responsabilidades del sistema para posteriormente organizar el trabajo.

Figura 7. Modelo 4+1 de Kruchten

Fuente. Elaboración personal basado en Architectural Language[34]

El otro tipo de decisiones corresponden al conjunto de decisiones de bajo nivel, tiene que ver con temas relacionados a los aspectos de estructura y comportamiento de cada componente y conector identificado en el nivel superior. Estas decisiones pueden incluir: la forma de organizar datos y manipularlos, también conocidas como de *estructuras de datos* o colecciones, la solución a problemas específicos reconocidos o *patrones de diseño*, *algoritmos especializados* que son codificados en el componente o conector para llevar a cabo su tarea, unidades discretas de trabajo con interfaces bien definidas como es el caso de los *servicios*, entre muchas otras decisiones.

En programación una *estructura de datos* es un componente especial que utiliza una forma particular de organizar y manipular datos en una computadora, con el fin de que pueda ser utilizado de manera eficiente[35] en otros algoritmos. Diferentes tipos de estructuras de datos son adecuados para

[34] MALAVOLTA, Ivanno. "Introduction to ARCHITECTURAL LANGUAGES". *Slideshare.net*. N.p., 2014. Miercoles 5 de Oct. 2016.
[35] JOYANES AGUILAR, Luis, and ZAHONERO MARTINEZ, Ignacio. "Estructura de Datos: Algoritmos, abstracción y objetos." (1999).

diferentes tipos de aplicaciones, y en algunos casos son altamente especializadas para tareas concretas.

Un *servicio* es una tecnología que utiliza un conjunto de protocolos[36] y estándares para gestionar una colección de recursos relacionados y expone su funcionalidad a través de interfaces a usuarios y aplicaciones[37]. Distintas aplicaciones de software desarrolladas en lenguajes de programación diferentes, y ejecutadas sobre cualquier plataforma, pueden utilizar los servicios para intercambiar datos o realizar operaciones especializadas en redes de ordenadores.

Los *patrones de diseño* son la base para la búsqueda de soluciones a problemas comunes en el desarrollo de software y otros ámbitos referentes al diseño[38]. Para que una solución sea considerada un patrón debe poseer ciertas características; una de ellas es que debe haber comprobado su *efectividad* resolviendo problemas similares en ocasiones anteriores, otra característica es la solución debe ser *reutilizable*, lo que significa que es aplicable a diferentes problemas de diseño en distintas circunstancias.

Según el nivel de abstracción utilizado para diseñar se distinguen las siguientes categorías de patrones de diseño (estas serán tratadas con mayor detalle en una sección posterior):

- *Patrón de arquitectura*: es una colección específica y nombrada de decisiones aplicables a problemas de diseño recurrentes, permiten

[36] Protocolo de comunicaciones. (2016, 26 de junio). Wikipedia, La enciclopedia libre. Fecha de consulta: 15:21, julio 30, 2016

[37] COULOURIS George, DOLLIMORE Jean, KINDBERG Tim. Sistemas distribuidos. Conceptos y diseño. Addison Wesley, 2001, pagina 8.

[38] FREEMAN Eric, FREEMAN Elisabeth, SIERRA Kathy, BATES Bert. Head First Design Patterns. O'Reilly Media, Inc, 2004.

organizar ciertas clases de sistemas software, más específicamente subsistemas[39].

- *Patrón de diseño*: son aquellos que expresan esquemas para definir estructuras de diseño (o sus relaciones) con las que construir sistemas de software.
- *Dialectos*: son patrones de bajo nivel específicos para un lenguaje de programación o entorno concreto.

También es importante mencionar el concepto de "*anti-patrón de diseño*", este tiene forma similar a la de un patrón de diseño, la idea de los anti-patrones es prevenir contra errores comunes y recurrentes de diseño en el software.

3.1 TÉCNICAS DE DISEÑO

El objetivo de esta sección es presentar un conjunto de técnicas organizadas que facilitan la actividad de diseño de software, vale la pena recordar que algunas técnicas mencionadas en secciones previas: *la separación de preocupaciones*, *la abstracción*, *la modularidad*, *el diseño descendente*, y también otro "conjunto de principios de ingeniería de software", tales como, *la anticipación del cambio* y *el diseño para generalidad*, en sí mismas, estas técnicas y principios proveen una guía al diseñador, sin embargo, es la experiencia del diseñador la que finalmente provee la guía necesaria para el uso de dichas herramientas de una forma efectiva.

3.1.1 Diseño funcional descendente[40]

Esta categoría cuenta con varias técnicas de diseño cuya esencia es realizar una descomposición gradual del sistema en elementos de menor complejidad,

[39] TAYLOR, Richard N. y MEDVIDOVIC, Nenad y DASHOFY, Eric M. Software Architecture. Foundations, Theory and Practice. Páginas 73. Wiley, 2010

[40] Esta sección se basa en las notas del curso de ingeniería de software publicadas en la siguiente fuente. "ETSI Informática, UNED". *ISSI-UNED*. N.p., 2016. Web. 26 May 2016.

de esta manera se logra una mayor y más fácil comprensión de la estructura y comportamiento de cada componente identificado y del sistema como un todo.

Una primera técnica desarrollada por Nicklaus Wirth en la década de los 70's se conoce como ***refinamiento progresivo*** consiste en plantear la aplicación como única operación global, y descomponerla gradualmente en operaciones más sencillas, detalladas y específicas. En cada nivel de refinamiento, se identifican operaciones, y se asignan a módulos separados, con límites claramente definidos. Esta técnica reduce el sistema hasta la utilización sólo de estructuras de control simples, con esquemas de ejecución con un único punto de inicio y un único punto de terminación.

Una segunda técnica desarrollada por Michael A. Jackson, es la conocida como ***diseño estructurado de Jackson***. Esta técnica hace uso de dos descripciones que toman la representación de diagramas. La primera descripción es el *diagrama de entidad-estructura*, el objetivo de este es encontrar una estructura del programa que se ajuste a las estructuras de datos de entrada y salida. Por otro lado, el *diagrama de especificación del sistema* es básicamente un diagrama de red que identifica las interacciones entre las entidades que constituyen el modelo del sistema, tales interacciones toman lugar mediante dos mecanismos de comunicación el *flujo de datos* y el *vector de estado*. El análisis de flujos de información realizado en la especificación del sistema consiste en identificar un flujo global de información desde los elementos de entrada hasta los elementos de salida.

También en esta categoría se tiene el ***diseño estructurado de Yourdon***. Se utilizan los diagramas de flujo de datos (DFD) para la descripción del funcionamiento de la aplicación. Con los primeros niveles de la descomposición obtenidos en el diagrama de flujo de datos (DFD), se construye un único diagrama en el que se incluyen los procesos implicados prescindiendo de los elementos de almacenamiento. Para este paso, se realiza

un análisis del DFD obtenido para de identificar bien sea, un único flujo global (flujo de transformación) o bien, uno o varios puntos en los que el flujo global se bifurca (flujo de transacciones, acciones realizados de forma paralela por el sistema). Finalmente, dado el patrón de comportamiento identificado del análisis anterior, se construye el diagrama de diseño mediante la notación de los diagramas de estructura. Para ello, a partir de los diagramas de flujo iniciales, se asignan procesos o grupos de procesos a los módulos de diseño y se establece una jerarquía o estructura de control (transformación o transacción).

3.1.2 Diseño con abstracciones[41]

En esta categoría se cuenta con la *técnica de* ***descomposición modular con abstracciones****,* consiste en identificar módulos dedicados a la administración de cada tipo abstracto de datos y de cada función importante dentro del sistema. Se emplea la notación de los *diagramas de bloques jerarquizados*; en los que se representan las relaciones de uso y donde la jerarquía se establece de arriba hacia abajo, es decir, los módulos de niveles superiores usan a los módulos de niveles inferiores. Se puede aplicar de forma descendente (*ampliación del refinamiento progresivo* con las abstracciones) o ascendente (*ampliación de primitivas* hacia abstracciones de nivel superior).

Otra técnica para esta categoría es el ***método de Abbott,*** este identifica los elementos abstractos que formarán parte del diseño, por lo regular extraídos de descripciones o especificaciones hechas en lenguaje natural, así de esta manera, los tipos aparecen como sustantivos, las operaciones como verbos y algunos adjetivos pueden sugerir atributos y un análisis cuidadoso de estos puede ayudar a establecer un rango de valores deseables para estos. Estas abstracciones se complementan y se organizan en un diagrama de diseño,

[41] Sección basada en. GOMEZ Sebastian, MORALEDA Eduardo. Aproximación a la ingeniería del software. Página 179-181. Editorial Universitaria Ramon Areces, 2014.

donde se asigna un módulo a una o un grupo de abstracción de datos relacionadas entre sí.

3.1.3 Diseño orientado a objetos

En esencia es similar a la técnica de diseño con abstracciones, pero además de las relaciones de uso y agregación, hay que considerar un nuevo tipo de relación conocida como *herencia* y un nuevo conjunto de conceptos que facilitan la descomposición modular del sistema tales como: *interfaces, polimorfismo, referencias, etc*. Cada módulo contiene la descripción de una o varias *clases de objetos* relacionados entre sí. Se suele utilizar un diagrama de estructura para describir la arquitectura del sistema y las relaciones de uso, y diagramas ampliados de las clases y de los objetos (instancias de clase) en las que se representan las distintas relaciones entre los datos y las funcionalidades.

Las técnicas de diseño actuales no se reducen a las pocas aquí mencionadas, sin embargo, estas dan una idea general de los aspectos comunes que comparten, *descomposición, nivel de abstracción, diseño modular, delegación*, son solo algunos de estos.

3.2 FUNDAMENTOS DE ARQUITECTURA[42]

La arquitectura de un sistema es "*el conjunto de conceptos fundamentales o propiedades de un sistema en su entorno, incluye: sus elementos, relaciones, y los principios de su diseño y evolución*".

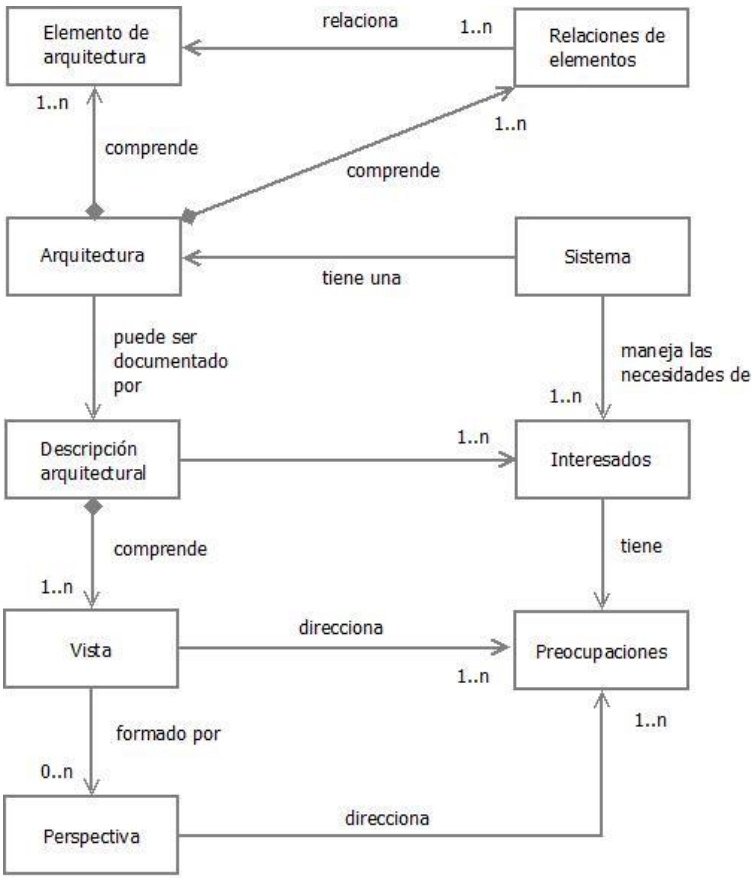
Los elementos que constituyen un sistema y las relaciones entre ellos definen la *estructura de alto nivel* del sistema, también conocida como *arquitectura*.

[42] Sección basada principalmente en: ROZANSKI Nick, WOODS Eoin. Software System Architecture. Addison Wesley, 2012.

Las estructuras, conectores y configuraciones[43] estáticas definen el diseño interno en el tiempo de desarrollo, mientras las estructuras dinámicas, configuraciones e interacciones definen el diseño en tiempo de ejecución.

Las propiedades fundamentales de un sistema se manifiestan de dos maneras diferentes: *comportamiento visible* (lo que el sistema hace) y las *propiedades de calidad* (cómo el sistema lo hace).

Figura 8. Conceptos núcleo de la arquitectura



Fuente. Software System Architecture44.

[43] Una configuración es una organización física o lógica de un conjunto de elementos determinados.
[44] ROZANSKI Nick, WOODS Eoin. Software System Architecture. Página 57. Addison Wesley, 2012.

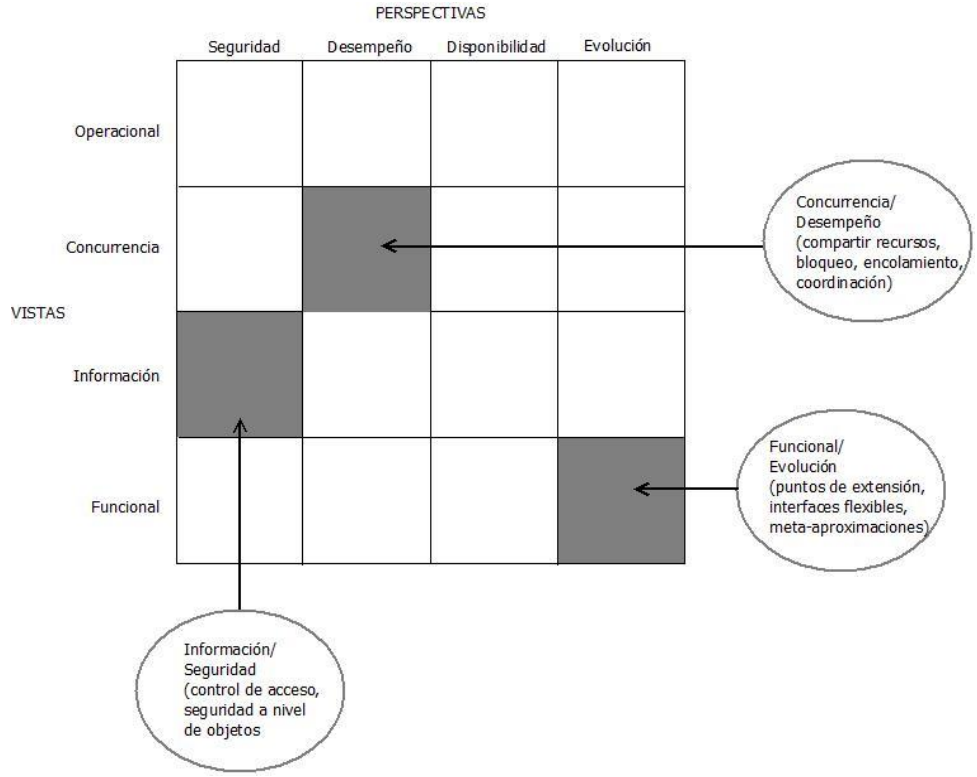
Durante el proceso de identificación de la arquitectura se crea lo que se conoce como una *arquitectura candidata* para el sistema, este es un arreglo particular de estructuras estáticas y dinámicas que tiene el *potencial* para exhibir los comportamientos visibles externamente y propiedades de calidad requeridos del sistema.

Por premisa todos los sistemas, sin importar su tamaño o complejidad, tienen una arquitectura, esto hace necesario tener una manera clara de representación para ésta, durante el proceso de desarrollo se puede emplear una *descripción arquitectural*, que consiste en un conjunto productos que documentan la arquitectura, facilitando que los interesados puedan entender y evidenciar que la arquitectura conduce de forma apropiada sus preocupaciones y requerimientos. En pocas palabras, una buena *descripción arquitectural* es aquella que comunica consistentemente los aspectos clave de la arquitectura a los interesados.

La descripción de un sistema complejo es más precisa si se emplea un conjunto de vistas interrelacionadas, las cuales pueden ilustrar de manera colectiva sus características funcionales y propiedades de calidad. Una *vista* es una representación de uno o más aspectos estructurales de una arquitectura que muestran cómo el sistema maneja las preocupaciones de cada uno de los interesados.

Por otro lado aunque el mecanismo de las vistas se basa en la separación de preocupaciones, hay aspectos que son transversales al conjunto de vistas y que no es práctico que sean modeladas con este mecanismo, para ello se emplea un conjunto de actividades, tácticas y guías que garantizan que el sistema exhibe un conjunto particular de propiedades de calidad que requieren consideración a lo largo de numerosas vistas de la arquitectura, lo anterior se resume en un mecanismo conocido como *perspectiva*.

Figura 9. Ejemplo de aplicar perspectivas a vistas



Fuente. Software System Architecture[45]*.*

Un ejemplo de un aspecto transversal al sistema (operacional, concurrencia, información y funcional), es la perspectiva de seguridad. Al aplicarla al sistema, esta no resulta en una nueva vista, pero permite identificar qué modificaciones deben ser hechas en cada vista para aportar a una mejor calidad de este aspecto. Así, si consideramos la seguridad en la vista de *Información* se puede adicionar elementos como *control de acceso y cifrado de datos*, si la consideramos en una vista funcional se puede adicionar elementos como autorización mediante *perfiles* y *permisos*.

[45] ROZANSKI Nick, WOODS Eoin. Software System Architecture. Pagina 53. Addison Wesley, 2012.

3.3 ESTILOS ARQUITECTÓNICOS VS PATRONES DE DISEÑO[46]

"*El diseño de la arquitectura define la relación entre los elementos estructurales mayores del sistema, los estilos y los patrones de diseño que pueden ser usados para alcanzar los requerimientos definidos por el sistema, y las restricciones que afectan la manera en la cual pueden ser implementados*" (Garlan y Shaw, 1996).

Los arquitectos y diseñadores traducen y mapean los requerimientos de software dentro de la arquitectura, convirtiéndola en el *diseño estratégico* frontal del sistema. Durante la etapa de definición de arquitectura, un diseñador debe especificar los elementos accesibles por los usuarios y las interconexiones que son visibles a los interesados. Por su lado, el diseño detallado, también llamado *diseño táctico*, se preocupa por las restricciones de diseño locales y los detalles internos de cada elemento constitutivo. Para aclarar lo anterior, considere el siguiente escenario de diseño, en la definición de la arquitectura de un sistema de control de tráfico, el diseñador puede especificar una *cola de prioridad* que almacene o despache las peticiones entrantes. En el diseño detallado, el diseñador debe elegir la estructura de datos de las alternativas de solución, por ejemplo, una cola de prioridad puede ser implementada usando una *lista simplemente enlazada*, una *lista doblemente enlazada* e incluso un *arreglo*.

La *arquitectura* provee los planos y una guía para desarrollar un sistema software basado en el análisis de la especificación de requerimientos. Una especificación completa de arquitectura de software debe describir no sólo los elementos y los conectores entre ellos, sino también las restricciones y comportamientos en tiempo de ejecución, de esta manera los desarrolladores saben qué y cómo el diseño debería ser implementado.

[46] Sección basada en: QIAN Kai, FU Xian, TAO Lixin, XU Chong-Wei, Jorge L.Díaz-Herrera. Software Architecture and Design Illuminated. Jones and Bartlett, 2010.

En la práctica, los diseñadores identifican *estilos de arquitectura* separando características comunes de elementos y conectores en "*familias de arquitectura*". Un *estilo arquitectónico* abstrae las propiedades comunes de una familia de diseños similares, y contiene un conjunto de reglas, restricciones semánticas y comportamientos relacionados cómo: la transferencia de control entre los elementos en el sistema, el equilibrio de los atributos de calidad, y patrones de cómo estructurar un sistema en un conjunto de elementos y conectores.

Un estilo arquitectónico se puede definir como una *n-tupla* de la forma:

$$\Phi = \{\epsilon, \eta, \theta\}$$

Donde:

- Un conjunto ϵ de componentes y conectores del sistema
- Un conjunto η de restricciones que definen cómo los elementos pueden ser integrados para dar forma al sistema
- Un conjunto θ de atributos que describen las ventajas y desventajas del estilo

Los *estilos arquitectónicos* son importantes debido a que cada uno fomenta un subconjunto de atributos de calidad, con esto se puede verificar si la arquitectura es consistente con la especificación de requerimientos, e identificar cuáles tácticas pueden ser usadas para mejorar la implementación de la arquitectura. Hay muchos estilos arquitectónicos de los cuales elegir, no es el objetivo de este documento dar una especificación detallada de cada uno, pero la siguiente lista ilustra algunos ejemplos.

1. Estilos de flujo de datos

- *Batch secuencial*: en este estilo el sistema se organiza como una secuencia de módulos que transforman lotes de datos de forma atómica, esto implica que un módulo se activa hasta que los módulos

precedentes completen su computación.

- *Tuberías y filtros*: este estilo descompone el sistema en componentes de fuentes de datos, filtros, tuberías y sumideros de datos. Las conexiones entre los componentes son *streams* (flujos) lo cual permite que varios componentes estén activos de forma concurrente incrementando la capacidad de ejecución.

2. Estilos centrados en datos

- *Repositorio*: en este estilo de arquitectura hay dos componentes principales: una estructura de datos pasiva[47] que representa el estado actual y una colección de componentes independientes que operan sobre él. En este caso la lógica de control y flujo es llevada a cabo por los procesos en los componentes y ellos afectan con acciones de recuperación y almacenamiento a la estructura de datos.
- *Blackboard*: en este estilo de arquitectura hay dos componentes principales: una estructura de datos que representa el estado actual y una colección de componentes independientes que operan sobre él. Si el estado actual de la estructura de datos dispara los procesos a ejecutar, el repositorio es lo que se llama una pizarra o un tablero de control. Es decir, la estructura de datos es activa.

3. Estilos Jerárquico

- *Main/Subrutina*: en este estilo el sistema se descompone en subrutinas invocadas jerárquicamente. El principal propósito es reutilizar y mantener subrutinas individuales desarrolladas independientemente.
- *Maestro/Esclavo*: este es una variante del estilo anterior, pero soporta tolerancia a fallos y adiciona un nivel de confiabilidad. En este estilo el componente esclavo provee servicios replicados al maestro y este a su vez elige un resultado entre los obtenidos por los esclavos. Este estilo

[47] Nota aclaración. Significa que el control que modifica el estado de la estructura de datos viene de las acciones realizadas de manera independiente por cada componente.

tiene la flexibilidad para que los esclavos realicen la misma tarea con diferentes algoritmos, o tareas completamente diferentes.

4. Estilos de interacción

- *Basado en eventos (sin buffer[48])*: este estilo separa el sistema en dos particiones, una fuente de eventos y un escuchador de eventos. El proceso de registro de eventos se encarga de conectar estas dos particiones. Es importante resaltar que no hay un buffer entre ambas partes.
- *Paso de mensajes (con buffer)*: este estilo separa el sistema en tres particiones: productores de mensajes, consumidores de mensajes y proveedores de servicio de mensajería. Estos se conectan asíncronamente por colas de mensajes o tópicos de mensajes.
- *Model View Controller (MVC)*: este estilo se basa en la separación en objetos que asumen un estereotipo: el **modelo** - es la estructura de información del dominio específico de la aplicación, la **vista -** maneja todo el aspecto gráfico utilizan los datos del modelo para su presentación y los **controladores** controlan las acciones entre la vista y el modelo.

5. Distribución

- *Cliente/Servidor*: es un modelo distribuido basado en la separación del sistema en dos procesos, usualmente ejecutan en procesadores diferentes que descompone el sistema en dos subsistemas. El cliente envía peticiones al segundo proceso y este las maneja y las responde.
- *Service Oriented Architecture (SOA)*: en este estilo el sistema se descompone en bloques de construcción llamados servicios. Un servicio es una unidad auto-contenida, bien definida de una funcionalidad del negocio. Los servicios pueden ser usados por otros servicios o por

[48] Definición. Memoria de almacenamiento temporal de información que permite transferir los datos entre unidades funcionales con características de transferencia diferentes.

lenguajes de orquestación de control de flujo.

6. Componente

- *Basada en componente*: en este estilo el problema es dividido en sub-problemas y cada uno es asociado con un paquete de software desplegable que recibe el nombre de componente. Un componente es un conjunto de funcionalidades reusable, reemplazable y autocontenido con una interfaz publica bien definida. (Se dará un detalle mayor en una sección posterior).

Por su parte los *patrones de diseño*[49] son modelos de trabajo enfocados a dividir un problema en partes de modo que sea posible abordar cada una de ellas por separado para simplificar su resolución. Los patrones de diseño comenzaron a reconocerse a partir de descripciones de varios autores a principios de 1990. Una definición más clara sobre qué es un patrón de diseño puede ser:

Un patrón de diseño es un conjunto de reglas que describen cómo afrontar tareas y solucionar problemas que surgen durante el desarrollo de software.

Tradicionalmente se dividen la mayoría de los patrones en tres conjuntos según su finalidad:

1. Patrones creacionales: definen modelos de creación de elementos para evitar una instanciación directa del elemento, proporciona a los programas una mayor flexibilidad para decidir qué objetos usar en una instancia del programa.

[49] A partir de este punto la sección se basa en apuntes extraídos de: FREEMAN Eric, FREEMAN Elisabeth, SIERRA Kathy, BATES Bert. Head First Design Patterns. O'Reilly Media, Inc, 2004

2. Patrones estructurales: como su nombre lo indica permiten determinar maneras de organizar colecciones de objetos para facilitar tareas complejas.
3. Patrones de comportamiento: permiten definir los mecanismos de comunicación entre los objetos y los flujos de información entre los mismos.

Si bien el estudio detallado de cada patrón queda fuera del alcance de este documento se describe a continuación algunos de los más popularmente utilizados para dar una mejor idea de en qué consisten.

1. Patrones creacionales

- *Singleton*: es un patrón de diseño diseñado para restringir la creación de objetos pertenecientes a una clase o el valor de un tipo a un único objeto. Su intención consiste en garantizar que una clase sólo tenga una instancia y proporcionar un punto de acceso global a ella.
- *Factoría:* es un patrón cuyo objetivo es devolver una instancia de múltiples tipos de objetos. Decide según ciertos parámetros analizados el tipo de objeto que devolver.
- *Constructor*: es usado para permitir la creación de una variedad de objetos complejos desde un objeto fuente (Producto), el objeto fuente se compone de una variedad de partes que contribuyen individualmente a la creación de cada objeto complejo.

2. Patrones estructurales

- *Adaptador*: este patrón permite transformar una interfaz de programación de una clase en otra.
- *Decorador*: este patrón responde a la necesidad de añadir dinámicamente funcionalidad a un Objeto. Esto nos permite no tener que crear sucesivas clases que hereden de la primera incorporando la nueva funcionalidad, sino otras que la implementan y se asocian a la primera.

- *Proxy*: es un patrón que permite cambiar un componente completo por otro. Tiene la ventaja de retardar la instanciación del componente requerido hasta que sea realmente necesario.

3. Patrones de comportamiento

- *Observador*: es un patrón de diseño que define una dependencia del tipo uno-a-muchos entre objetos, de manera que cuando uno de los objetos cambia su estado, notifica este cambio a todos los dependientes.
- *Visitante*: Es un patrón de comportamiento, que permite definir una operación sobre objetos de una jerarquía de clases sin modificar las clases sobre las que opera.
- *Iterador*: Este patrón tiene como función permitir recorrer un conjunto de datos mediante una interfaz estándar sin tener que conocer los detalles de la implementación de los datos.

Tanto *estilos de arquitectura* como *patrones de diseño* ofrecen mecanismos avanzados para resolver problemas, pero ambos trabajan en niveles de abstracción diferentes y no son mutuamente excluyentes, sino más bien complementarios. Es necesario resaltar su importancia en el desarrollo del aplicativo, ya que ellos representan un estado *prescriptivo* del diseño de la aplicación.

3.4 ATRIBUTOS DE CALIDAD[50]

Cada estilo arquitectónico tiene sus ventajas, desventajas y riesgos potenciales. Elegir el estilo correcto para satisfacer las funciones requeridas impacta también el set de atributos de calidad del sistema. Los *atributos de*

[50] Sección basada en: GORTON, Ian. Essential Software Architecture. Páginas 23-39, Springer-Verlag Berlin Heidelberg 2006.

calidad son identificados en el proceso de análisis de requerimientos, estos pueden catalogarse en tres grupos:

1. **Atributos de implementación** (no observables en tiempo de ejecución)

    - *Interoperabilidad*: accesibilidad universal y la habilidad de intercambiar datos entre componentes internos y con el mundo externo.
    - *Mantenibilidad y extensibilidad*: la habilidad para modificar el sistema y extenderlo convenientemente.
    - *Capacidad de prueba*: el grado para el cual es sistema facilita el establecimiento de casos de prueba, usualmente requiere un conjunto completo de documentación acompañado del diseño del sistema y la implementación.
    - *Portabilidad*: el nivel de independencia del sistema de las plataformas software y hardware.
    - *Escalabilidad*: es la habilidad del sistema para adaptarse a un incremento en las peticiones de usuario.
    - *Flexibilidad*: la facilidad de modificación del sistema para ajustarse a diferentes entornos o problemas para los que el sistema no fue originalmente diseñado.

2. **Atributos en tiempo de ejecución**

    - *Disponibilidad*: la capacidad de un sistema de estar disponible 7/24.
    - *Seguridad*: habilidad de un sistema para manejar ataques maliciosos de fuera o del interior del sistema.
    - *Rendimiento*: incremento de la eficiencia de un sistema con consideraciones de tiempo de respuesta, throughput, y utilización de recursos atributos usualmente en conflicto con otros.
    - *Usabilidad*: el nivel de satisfacción humana de usar el sistema. La usabilidad incluye completitud, correcto funcionamiento, compatibilidad,

también como una interfaz de usuario amigable, documentación completa y soporte técnico.

- *Mantenibilidad (extensibilidad, adaptabilidad, compatibilidad y configurabilidad)*: cambios fáciles al sistema software.

3. **Atributos del negocio**

- *Tiempo de mercado*: el tiempo tomado desde el análisis de un requisito hasta la fecha que el producto es liberado.
- *Costo*: el valor de construir, mantener y operar el sistema.
- *Tiempo de vida*: el periodo de tiempo que el producto está "vivo" antes de ser retirado.

En la mayoría de los casos, un estilo de arquitectura no puede soportar todos los atributos de calidad simultáneamente. Los arquitectos de software con frecuencia necesitan *balancear valores entre atributos opuestos*. Algunos *pares de compensación* entre atributos de calidad incluyen:

- *Compensación entre espacio y tiempo*: por ejemplo, para incrementar la eficiencia de una tabla hash implica un decremento en su eficiencia en espacio.
- *Compensación entre confiabilidad y rendimiento*: por ejemplo, los programas java están bien protegidos contra desbordamiento de buffer debido a medidas de seguridad tales como chequeo de límites sobre arreglos, esto implica más procesamiento.
- *Compensación entre escalabilidad y rendimiento*: por ejemplo, una aproximación típica para incrementar la escalabilidad de un servicio es la replicación de servidores. Para asegurar la consistencia de todos los servidores el rendimiento del servicio completo se ve comprometido.

Cuando un estilo arquitectónico no satisface todos los atributos de calidad deseados. El arquitecto de software trabaja con el analista del sistema y los

interesados para priorizar los atributos de calidad. Al realizar diseños de arquitectura alternativos y el calcular los pesos de evaluación de los atributos de calidad, los arquitectos de software pueden seleccionar un diseño "*más óptimo*".

3.5 COMPONENTES SOFTWARE

Como se mencionó, una *arquitectura de software basada en componentes* divide el problema en sub-problemas y asociada cada división con un componente con límites bien definidos. La razón fundamental para incluir esta pequeña sección se relaciona con el tema de los atributos de calidad que favorecen este estilo de arquitectura, entre los que cabe mencionar: reusabilidad, productividad, adaptabilidad, escalabilidad.

¿Qué es entonces un componente? Un componente es un conjunto de funcionalidades modulares, desplegables, reemplazable que encapsula su implementación y expone interfaces de alto nivel. Las interfaces del componente juegan un rol importante en este tipo de diseño, dado que permiten realizar conexiones entre sí, constituyendo el sistema como una red *interconectada de componentes*, donde cada uno, expone y requiere un conjunto de servicios, los cuales pueden ser síncronos o asíncronos, y ser conectados mediante flujos de entrada/salida.

La principal motivación detrás de este diseño es la reusabilidad. Ya hace mucho se vienen explotando esta ventaja y prueba de ello son los muchos marcos de trabajo estándar basados en esta arquitectura, tales como *JavaBean, EJB, CORBA, .NET web services, y grid de servicios*.

Un *diseño orientado a componentes* representa un nivel más alto de abstracción que un *diseño orientado a objetos equivalente*; el diseñador define componentes y conexiones entre ellos en lugar de clases y conexiones entre

clases. En esencia un componente es un concepto de más alto nivel, usualmente encapsulando una colección de clases. Así, en diseño orientado a componentes primero se identifican los componentes y sus interfaces en lugar de identificar clases y sus relaciones. El principal reto es diseñar componentes de manera tal que se haga posible adaptar componentes existentes para reusarlos.

En pocas palabras, cada componente puede ser visto como una caja negra que agrupa funcionalidades y datos cohesivamente como un paquete.

3.6 MIDDLEWARE[51]

Para introducir el concepto de Middleware, es recomendable presentar primero un par de definiciones de distintos autores para disponer de un panorama aproximado sobre en qué consiste:

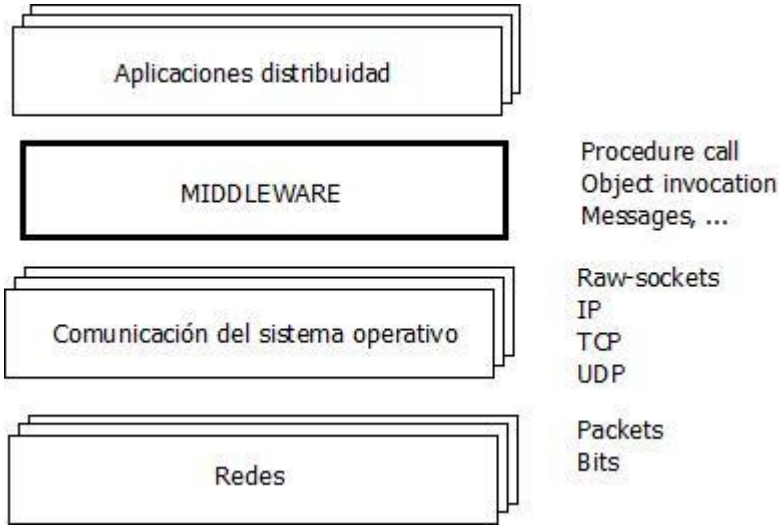
Definición 1[52]: "El Middleware es una capa de un sistema distribuido que ofrece un conjunto de servicios que hacen posible el funcionamiento de aplicaciones distribuidas sobre plataformas heterogéneas. Funciona como una capa de abstracción de software distribuida, que se sitúa entre las capas de aplicaciones y las capas inferiores (sistema operativo y red)."

Definición 2[53]: "Middleware es un software que permite conectar componentes o aplicaciones. Consiste en un conjunto de servicios que permiten que múltiples procesos corriendo en una o varias máquinas interactúen de un lado a otro de la red."

[51] Sección basada en: COULOURIS George, DOLLIMORE Jean, KINDBERG Tim. Sistemas distribuidos. Conceptos y diseño. Addison Wesley, 2001.
[52] Qusay H. Mahmoud. Middleware for Communications. John Wiley & Sons, 2005.
[53] Luis Hernández Echeverría. (mayo 2015). Middleware. Julio 2016, de Blogger.com Sitio web: http://mov-middleware.blogspot.com.co/

Al proporcionar una capa de abstracción los detalles de implementación del proceso de comunicación entre aplicaciones en nodos de computación diferentes quedan encapsulados en dicha capa de software. Así los temas relacionados a serialización de mensajes, localización de servicios, encapsulación de mensajes, determinación del protocolo de transporte, construcción del mensaje de petición, interpretación, deserialización y publicación de la respuesta, son responsabilidad delegada a esta capa de software intermedio.

Figura 10. Capa de middleware y relación con el sistema



Fuente. Sistemas distribuidos. Conceptos y diseño[54]

El middleware brinda la abstracción de la complejidad y heterogeneidad tanto de las redes de comunicaciones subyacentes como de los sistemas operativos y lenguajes de programación, proporcionando así, una API para acceder a una fácil programación y manejo de aplicaciones distribuidas.

Sin esta capa mediadora entre aplicación y sistema operativo, la programación de aplicaciones distribuidas no se haría práctica debido al extenuante trabajo

[54] COULOURIS George, DOLLIMORE Jean, KINDBERG Tim. Sistemas distribuidos. Conceptos y diseño. Addison Wesley, 2001.

de programación con API’s de bajo nivel que tendrían que lidiar con puntos de conexión (Sockets) y con todos sus flujos de excepción. Esta capa representa una gran ventaja al proveer soporte para temas de sincronía entre procesos, concurrencia, transaccionalidad, control de acceso. Además, ofrece una gran flexibilidad para ser considerada relaciona en temas de adaptación.

3.7 CONCLUSIONES

- Este capítulo brinda al lector un panorama general sobre los conceptos de diseño y en particular de la arquitectura de software, sentado las bases sobre las cuales se desarrollará el marco de trabajo.
- El diseño de software puede dividirse en dos niveles el diseño estratégico y el diseño táctico. Esto permite realizar explícitamente una separación de responsabilidades y adoptar mecanismos de adaptación adecuados al tipo del nivel considerado.

4. INGENIERÍA DE DESARROLLO ADAPTATIVO

Mientras realizar diseños para el cambio y la encapsulación son principios básicos e importantes para soportar un proceso de adaptación, la complejidad de los actuales sistemas requiere de mecanismos más poderosos para soportar comportamientos de adaptación, estos mecanismos pueden incluir:

- *Marcos de trabajo* que tienen por finalidad abarcar y gobernar el proceso completo de adaptación.
- *Estilos arquitectónicos* que soporten cambios a gran escala del sistema.
- *Técnicas específicas* que asistan la mayor parte de las tareas de adaptación: monitoreo, análisis, planeación y ejecución.

El principal objetivo de este capítulo es introducir un amplio espectro de conceptos y mecanismos usados para soportar un proceso de adaptación y que serán la base de la *propuesta del marco de trabajo*, también se estudiarán algunos marcos de referencia, analizando tanto sus partes constituyentes como los principios que las dirigen y mecanismos que emplean. El objetivo principal de este capítulo es definir y delimitar claramente el ámbito de aplicación y la contribución del presente trabajo.

4.1 APROXIMACIONES DE ARQUITECTURAS PARA LA ADAPTACIÓN

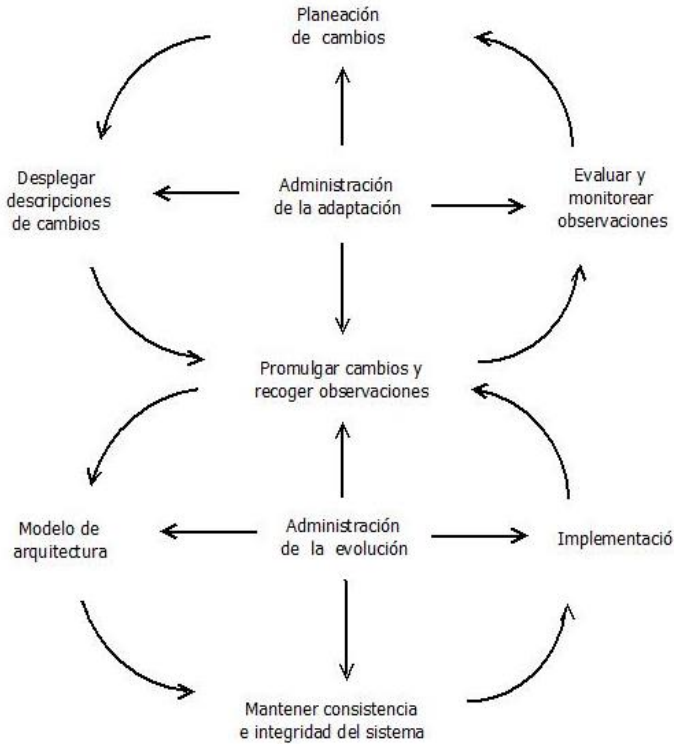
4.1.1 Aproximación conceptual[55]

La primera aproximación a un marco de trabajo conceptual basada en una arquitectura adaptativa fue presentada en 1999 (Oreizy et al. 1999). Este

[55] Sección basada en: Peyman Oreizy, Michael M. Gorlick, Richard N. Taylor, Dennis Heimbigner, Gregory Johnson Nenad Medvidovic, Alex Quilici, David S. Rosenblum, and Alexander L.Wolf. An architecture-based approach to self-adaptive software, IEEE Intelligent Systems. 1999.

marco de trabajo presentado en la imagen de la Figura 11. describe las actividades principales, las entidades y las relaciones clave entre estas.[56]

Figura 11. Arquitectura conceptual para la adaptación



Fuente. Software Architecture. Foundations, Theory and Practice[57]

El diagrama presenta una arquitectura dividida en dos ciclos, el ciclo en la parte superior etiquetada como "*Administración de la adaptación*" se enfoca en los mecanismos empleados para que el sistema responda a cambios en su entorno, dicho ciclo puede incluir humanos o puede ser por completo autónomo. Las etapas de este ciclo de adaptación son: el *monitoreo y evaluación de las observaciones,* este se refiere a los mecanismos empleados

[56] Un trabajo basado en esta arquitectura y recomendado al lector. Brice Morin, Thomas Ledoux, Mahmoud Ben Hassine, Franck Chauvel, Olivier Barais and Jean-Marc Jézéquel, Unifying Runtime Adaptation And Design Evolution, Institut de recherche en informatique et systèmes aléatoires, 2009. Donde se presenta un método para integrar Adaptación y evolución.

[57] Richard N. Taylor, Nenad Medvidovic, Eric M. Dashofy. Software Architecture. Foundations, Theory and Practice. Pagina 540. Wiley, 2010

para observar y evaluar el comportamiento de una aplicación en su entorno de ejecución, de los resultados de esta evaluación se crean descripciones detalladas para la introducción de cambios en el sistema, esto se conoce como la etapa de *planeación de cambios,* dichas descripciones tienen por meta crear un plan que facilite la ejecución de acciones de adaptación. En la siguiente parte de este ciclo está la etapa de *despliegue de descripción de cambios* donde se coordina la transmisión de acciones planeadas.

El ciclo inferior en la gráfica, etiquetada como "*Administración de la evolución*", se enfoca en los mecanismos que se emplean para cambiar la aplicación software de forma permanente. Como puede observarse en este ciclo inferior se parte de un *modelo de arquitectura*, algo así como, un esquema inicial a partir del cual se consideran los cambios a realizar, los cambios al modelo de arquitectura se reflejan en modificaciones para la *implementación* de la aplicación, mientras se asegura que el modelo y la implementación son consistentes uno con el otro.

Un aspecto crítico de la evolución de un sistema en ejecución es la administración del cambio, este tiene por responsabilidad identificar qué debe ser cambiado; y proveer el contexto sobre el cual razonar, especificando e implementando cambios; mientras garantiza la integridad del sistema.

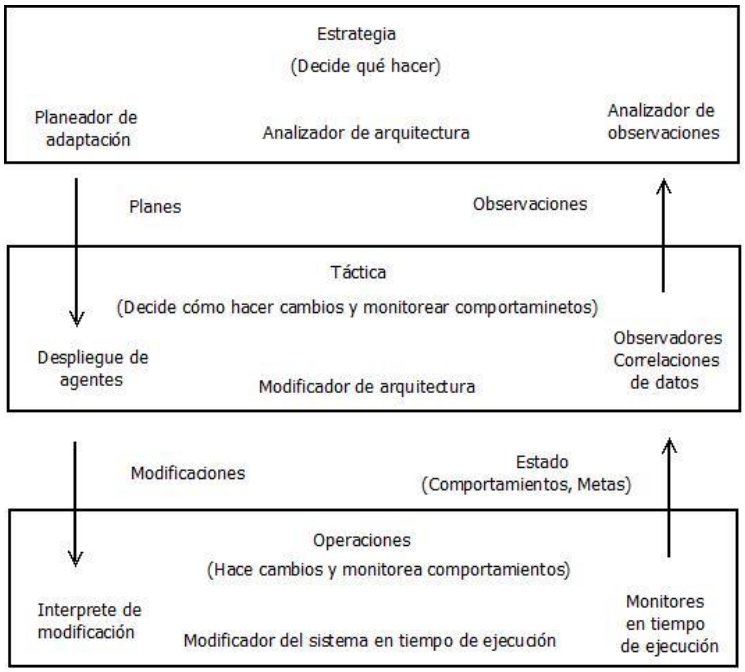
El punto de mediación entre ambos ciclos recibe el nombre de *difusión de cambios y recolección de observaciones* funciona a manera de canal de comunicación entre ambos, coordinando la separación de responsabilidades. El ciclo superior es quien determina y da forma al cambio, pero es el ciclo inferior quien lo ejecuta y lo lleva a cabo. Este *mediador* también puede recoger observaciones del efecto producto de las modificaciones previas, y realimentar el ciclo superior para contar con más información para la toma de decisiones.

Un punto de vista crítico al modelo revela cómo el objetivo de esta arquitectura es arraigar en el "*ADN*" de la aplicación los cambios identificados, es un modelo muy ambicioso que podría conducir la aplicación hacia un estado operativo óptimo, pero también puede llevar está a caer en estados no deseados e irreversibles. Por otro lado, la administración de la adaptación y el mantenimiento juega roles clave en esta aproximación, pues dada la naturaleza de la misma, los riesgos engendrados por modificaciones en tiempo de ejecución podrían pesar más que aquellos asociados con la detención y reinicio del sistema.

4.1.2 Aproximación basada en arquitectura adaptable

En este *marco de trabajo* se presentan actividades, entidades y operadores separados en tres grupos con una misión similar de la anterior aproximación, pero con información adicional detallada.

Figura 12. Arquitectura basada en adaptación



Fuente. Software Architecture. Foundations, Theory and Practice[58]

El sistema se divide en tres grupos, estos son: *estratégico, táctico y operativo*. La estrategia se refiere a determinar el *qué hacer,* la táctica se enfoca al desarrollo de un *plan detallado* para alcanzar las metas de la estrategia, es decir, *el cómo* y finalmente el operativo, constituye el nivel más bajo y se enfoca en los mecanismos técnicos para llevar a cabo el plan detallado.

En las actividades del *nivel de estrategia* se localizan las entidades que van a analizar información acerca de la arquitectura y del sistema en su entorno de operación, estas entidades pueden ser simplemente tareas que un humano realice, o pueden llegar a ser programas completamente autónomos.

En las actividades del *nivel táctico* se encuentran las entidades que realizan modificaciones específicas a la arquitectura. Estas entidades llevan a cabo su trabajo basados en el plano proporcionado por el *nivel estratégico* y en la información recogida por ciertas *entidades de despliegue* que observan el impacto de tales modificaciones. En este nivel también se encuentran entidades para determinar cómo correlacionar la información monitoreada recibida de las instancias desplegadas del sistema de manera tal que los datos recogidos sean confiables para el análisis en la capa superior.

Finalmente, en las actividades del *nivel operativo*, están todos los mecanismos específicos para modificar los modelos y la implementación de la arquitectura, una de sus principales metas es mantener la consistencia entre ambos elementos, además de proporcionar mecanismos para reunir datos de las instancias en ejecución.

[58] Richard N. Taylor, Nenad Medvidovic, Eric M. Dashofy. Software Architecture. Foundations, Theory and Practice. Pagina 541. Wiley, 2010

4.1.3 Aproximación basada en sistemas multi-agente[59]

Los sistemas complejos autonómicos que *no* están constituidos por componentes simples autoadministrables pueden ser construidos usando agentes "*inteligentes*".

Un *agente* es un sistema autónomo proactivo que posee habilidades sociales. Cada agente tiene sus propias metas, y estas dirigen sus decisiones. El reto en este tipo de aproximación es definir las metas individuales de manera tal que la meta global sea cumplida, y coordinar las acciones entre los distintos agentes.

En un sistema autónomo, se busca que metas relacionadas con nociones de alto nivel se puedan convertir en acciones concretas y que los agentes por sí mismos determinen cuál es el comportamiento necesario para alcanzarlas.

El modelo *top-down* propuesto de coordinación jerárquica para la aplicación de agentes consiste en dividir una tarea en pasos, y cada paso puede incluso ser dividido en sub-pasos. Un paso puede entonces ser asignado a un agente de ejecución, que conserva una agenda de tareas a completar. En esencia sigue los principios de diseño del *refinamiento progresivo*, explicado con detalle en el capítulo anterior. Aunque en sistemas de múltiples agentes cada componente exhibe su propio comportamiento autónomo, hay una clara separación entre el componente convencional que realiza una tarea y el administrador autónomo el cual implementa autoadministración sobre este. Sin embargo, en algunos sistemas el componente autónomo es inseparable de la lógica principal de la aplicación en el agente.

Al comparar esta aproximación con las arquitecturas basadas en diseño, salta a la vista que el sistema multi-agente adaptativo puede tener una arquitectura

[59] Sección basada en: TESAURO Gerald, CHESS David M., WALSH William E. A Multi-agent System Approach to Autonomic Computing. IBM TJ. Watson Research Center. 2004

distribuida. Sin una estructura de monitoreo centralizada, los agentes deben monitorearse a sí mismos (*monitor interno*) pero también a otros agentes (monitor externo). El monitoreo externo puede lograrse proactivamente mediante el envío del *heartbeat* o *pulso* de un agente regularmente sobre un canal de broadcast que otros agentes envían y escuchan. El *heartbeat* provee un resumen del estado de un agente a otros agentes responsables que monitorean dicho estado.

Antes de abordar con mayor detalle el tema de los sistemas multi-agente para sistemas adaptativos, se describen algunas técnicas mínimas a ser consideradas para el ciclo de vida de adaptación, explicando en cada etapa el tipo de consideraciones para tener en cuenta.

4.2 TÉCNICAS EMPLEADAS EN EL CICLO DE VIDA DE ADAPTACIÓN[60]

Las siguientes secciones se enfocan en presentar diversas técnicas que pueden ser usadas en cada etapa del ciclo de vida de un sistema adaptativo, estas podrían utilizarse en el diseño del marco de trabajo para adaptación. Se recuerda al lector que la descripción dada es básica, pero se vincula con referencias bibliográficas para información extra, el objetivo es mostrar un panorama general de estas técnicas dentro del proceso de adaptación.

4.2.1 Técnicas de observación y captura del estado

Una adaptación es activada cuando alguien (un usuario, un desarrollador, un componente, etc.) determina que el comportamiento actual del sistema sufre una desviación respecto a los objetivos trazados para el mismo, así la observación se convierte en la piedra angular de las condiciones que inician el proceso de adaptación.

[60] Sección basada en: Richard N. Taylor, Nenad Medvidovic, Eric M. Dashofy. Software Architecture. Foundations, Theory and Practice. Wiley, 2010

Las observaciones se realizan sobre el estado del sistema, en este punto, es bueno acordar lo qué se entiende por *estado* en este contexto: "*el estado de un sistema es el conjunto de los valores en tiempo de ejecución de los objetos que lo constituyen"*. Como el lector comprenderá, estos valores están cambiando continuamente, y por lo tanto el estado debe considerarse un elemento dependiente del tiempo. Por otro lado, una mirada minuciosa respecto de la definición dada es que no se obliga a conocer el conjunto completo de los valores de todos los objetos del programa, por lo tanto, se puede tomar un subconjunto de los valores de los objetos del programa que sea representativo de su *estado*.

Si bien el *estado* es una aproximación a la descripción del funcionamiento del aplicativo, hay otra información que también resulta ser de importancia para el propósito de la adaptación, esta información proviene principalmente del exterior del sistema, es decir, del entorno de ejecución del programa. ¿Qué información debería formar parte del estado completo de un sistema? Una respuesta simple a esta pregunta es: *todo conjunto de datos del cual se puede extraer información valiosa sobre el estado del sistema*. Con base en esto, se puede adicionar a la información ya mencionada las especificaciones y modelos del sistema, de su diseño incluso su arquitectura. Las *especificaciones analizadas* pueden estar sujetas a metas explícitas, cambiantes y dinámicas para la aplicación. Una *arquitectura observada*, es la arquitectura de la aplicación como existe en su estado actual, es decir, su conjunto de componentes, conectores y las configuraciones entre ellos, que pueden ser cambiantes y dinámicas dependiendo de la aplicación.

Una vez acordada la definición de estado de un programa, se listan algunas técnicas empleadas para su monitoreo y captura del estado.

4.2.1.1 Técnicas relacionadas a la depuración de un programa.

En caso de requerir que el análisis no implique ninguna modificación en el software:

- *Análisis manual*: consiste en observar la interfaz gráfica de usuario del programa y realizar observaciones sobre los valores de su estado.
- O*bservación de código*: la inspección del código fuente de la aplicación cuando se dispone de él puede conducir a lecturas del estado estático del programa y conocimiento de la arquitectura.
- *Monitoreo automático*: habilita el programa para dar información en tiempo real de su estado y actividades en tiempo de ejecución sin requerir ninguna modificación a este.
- *Analizadores de hardware*: en algunos contextos, tal como *sistemas embebidos*, los analizadores de hardware pueden permitir la inspección del estado de un programa sin ninguna perturbación del software en su entorno en tiempo de ejecución.

4.2.1.2 Técnicas con acceso a modificación del software.

Si la modificación del software es factible, se puede utilizar una amplia variedad de técnicas más especializadas y objetivas:

- *Logger: consiste en* registrar de las actividades del sistema mediante sentencias de impresión especiales, un nivel primitivo, pero en ocasiones suficiente.
- *Monitores especializados:* consiste en insertar en el código de la aplicación entidades de monitoreo que funcionan como aserciones, es decir, chequean una condición definida, y si esta se satisface, se registra el valor o se genera una alarma de su ocurrencia. Este tipo de monitoreo debería permitir que la ejecución proceda sin obstáculos.
- *Monitoreo de interfaces:* consiste en enfocarse en la captura de información en los límites de un componente, al hacerlo se captura información que pase a través de los conectores de una aplicación, es

una buena técnica cuando se utilizan mecanismos de comunicación entre componentes como el middleware.

- *Modelo basado en eventos:* consiste en la implementación de tecnologías que definen la estructura de un evento en un formato consistente y común. Tiene la ventaja de permitir cambiar cosas más allá de las aparentes circunstancias iniciales, dado que cada evento encapsula un *mensaje de datos* como resultado de la ocurrencia de una situación presentada.
- *Protocolos de administración:* esta categoría se enfoca en aprovechar protocolos creados en la industria utilizando sus capacidades para la tarea del acopio del estado de la aplicación. Un ejemplo a nivel de protocolo puede ser SNMP (Simple Network Management Protocol) las ideas detrás de este protocolo para la red y sistemas distribuidos también son aplicables a los sistemas autoadaptables.
- *Estándares abiertos:* son estándares creados para monitoreo y diagnóstico en aplicaciones empresariales complejas, habilitan a los desarrolladores para crear sistemas administrados con la capacidad de medir la disponibilidad de la aplicación, rendimiento, uso y tiempo de respuesta, entre otras características.

En algunos contextos de adaptación, la técnica usada para reunir las observaciones también puede ser responsable de transmitirlas desde plataforma donde la aplicación observada está ejecutándose hacia el lugar en cual se realizará análisis de la adaptación. También es posible llevar a cabo combinación de estas técnicas, incluso aun cuando se refieran a niveles de abstracción diferentes.

4.2.2 Técnicas para analizar datos

Analizar los datos reunidos en el acopio del estado del sistema y consecuentemente determinar qué tipo de adaptaciones realizar, es un problema donde se ha realizado una amplia investigación. El principal

obstáculo de esta actividad es la diversidad en los *tipos de cambio*[61], que el sistema puede presentar. En la medida en que estos *tipos de cambio* puedan ser anticipados durante el diseño del sistema, las técnicas de observación antes mencionadas pueden ser localizadas en lugares adecuados del aplicativo para monitorear situaciones específicas, y desencadenar la ejecución de estrategias cuando los *monitores* detecten valores o tendencias clave del estado del sistema.

Una definición fundamental para entender el objetivo de estas técnicas es entender en qué consiste el *análisis de datos*, en su forma más simple debe entenderse cómo: "*la comparación de un valor observado con otro valor que representa un punto de referencia"*. Un ejemplo, consiste en revisar la carga de trabajo de una instancia de un componente de software, si el número de transacciones de los clientes conectados a este componente se incrementa fuera de sus límites máximos de operación (*punto de referencia*), dependiendo de ciertas políticas sería apropiado emitir una alarma ante esta situación.

Esta etapa se enfoca en ¿cómo detectar el cambio?, a continuación, se mencionan algunas de las técnicas:

- *Reglas de monitoreo:* la idea principal consiste en crear un conjunto de reglas para monitorear un conjunto de valores. Las condiciones en las reglas pueden ser muy elaboradas y complejas (es decir, considerar muchos tipos de observaciones y relaciones entre tales valores).
- *Comportamiento dirigido por un modelo:* la idea con esta técnica consiste en comparar el comportamiento observado de un sistema con el comportamiento *predefinido* su diseño previo, antes de su implementación.

[61] Nota de aclaración. Para comprender la connotación del término *tipo de cambio* en este contexto, se sugiere al lector la sección 2.5 del Capítulo 2 de este trabajo.

- *Árbol de decisión hard-code:* es una opción estática del proceso de decisión. Durante la etapa de diseño, se construye un *árbol de decisión*[62] que abarca las reglas para ejecutar los cambios, sin embargo, esto implica que la modificación de dicho árbol requiere compilar y desplegar de nuevo el sistema o algunos de sus componentes.
- *Basada en políticas o definiciones de calidad de servicio:* en esta aproximación las políticas son definidas y administradas externamente, así pueden ser cambiadas en tiempo de ejecución, y de esta manera las condiciones que sugieren cambios se ajustan a estas nuevas definiciones para crear comportamientos nuevos para requerimientos funcionales y no funcionales.

4.2.3 Técnicas para descripción de cambios

Para efectuar modificaciones a un sistema software en ejecución, los cambios deben ser especificados en descripciones comprensibles por la máquina. A continuación, se presentan maneras concretas para expresar los cambios en la arquitectura:

- *Scripts de cambios:* son programas ejecutables que operan en el sistema para hacer cambios. Estos scripts formarán parte del API (*application programming interface*) de adaptaciones subyacente proporcionado por el sistema implementado. Dicho API provee funciones de alto nivel, por ejemplo, instanciar y remover componentes y conectores, crear y destruir relaciones entre elementos, podría incluso gestionar los servicios proveídos.
- *Diferenciales arquitectónicos:* en lugar de programas ejecutables o scripts, los cambios de arquitectura pueden ser expresados como un conjunto de diferencias entre una arquitectura A y una arquitectura A'. Tales diferencias son en esencia una lista de adiciones, remociones y modificaciones de los elementos de la arquitectura. Así, se analiza una

[62] Es una técnica que permite analizar decisiones secuenciales basadas en el uso de resultados y probabilidades asociadas.

situación, se recupera la modificación de la lista de diferenciales y se emplea para modificar el sistema.

- *Nuevo modelo de arquitectura:* es un modelo de arquitectura completo usado para describir la arquitectura de adaptación objetivo. En este caso, un diferencial de arquitectura es creado examinando los elementos en el sistema actual, como también aquellos en el nuevo modelo y determinar que adicionar y qué remover para llevar el sistema actual hacia la nueva arquitectura. Funciona como un volcado de cambios completos sobre el sistema.

4.2.4 Técnicas para aplicación de cambios

Al considerar las técnicas de descripción de cambios: *scripts, diferenciales y nuevos modelos* y garantizar que estos sean efectivos en un escenario de adaptación real, la implementación del sistema desplegado y el modelo de arquitectura deben tener una correspondencia estrecha entre elementos. En este caso pueden usarse marcos de trabajo de implementación de arquitectura o middleware flexible para facilitar el trabajo.

Entre las técnicas para aplicación de cambios se tienen:

- *Humano como agente de cambio:* el artefacto de cambio es revisado por una persona, y esa persona es responsable por llevar a cabo las correspondientes modificaciones al sistema.
- *Equipo de consultoría como agente de cambio:* ciertos sistemas software son tan complejos para desplegar, instalar y configurar que los clientes simplemente no pueden hacerlo ellos mismos. En general, contratan un equipo de consultores o expertos del producto para realizar las modificaciones.
- *Scripts de cambio:* el fragmento binario es ejecutado en un entorno de ejecución localizado con la aplicación objetivo. El software subyacente de la aplicación objetivo, usualmente marcos de trabajo de implementación,

de arquitectura, o middleware, es invocado para realizar los cambios al sistema.

- *Diferenciales arquitectónicos:* un agente de cambio ejecutado a lo largo de la aplicación objetivo interpreta las diferencias y realiza los cambios en esta, en un proceso conocido como mezcla.
- *Nuevo modelo de arquitectura:* un agente de cambio debe determinar la diferencia entre el actual estado de la aplicación y la configuración de la aplicación destino y realizar todos cambios necesarios.
- *Programación orientada a aspectos:* tiene por intención permitir una adecuada estructura modular de las aplicaciones y posibilitar una mejor separación de responsabilidades, encapsula los diferentes conceptos que componen una aplicación en entidades bien definidas, de esta forma se consigue razonar mejor sobre los conceptos, se elimina la dispersión del código y las implementaciones resultan más adaptables y reusables.

4.2.5 Tipos de acciones de adaptación

- *Almacenar en caché:* almacenar datos, estados, conexiones, objetos o componentes en caché con el fin de disminuir el tiempo de respuesta, disminuir la carga del servidor, o ayudar a descentralizar la administración.
- *Almacenar datos de calidad en caché:* permite optimizar el uso de ancho de banda para algunas aplicaciones e incrementar la velocidad.
- *Comprimir datos* es una medida que ayuda a optimizar el uso del ancho de banda mediante la comprensión de datos
- *Ajuste de parámetros o refinamiento:* ajustar parámetros como por ejemplo el tamaño de un buffer o tiempo de retardo, con el ánimo de alcanzar ciertas metas de adaptación.
- *Balanceo de cargas:* consiste en realizar una división “justa” de la carga de trabajo entre los elementos del sistema para alcanzar una máxima utilización, rendimiento o minimizar el tiempo de respuesta.

- *Cambio de aspecto:* cambiar el aspecto de un componente u objeto por otro con una calidad diferente.
- *Cambio de algoritmo/método:* como su nombre lo indica la idea es cambiar un algoritmo particular satisfacer restricciones en tiempo de ejecución u otras propiedades deseables.
- *Adición de componentes:* permite aumentar las funcionalidades del sistema, no se debe asumir que el sistema se encuentra en su estado inicial, el componente adicionado debe descubrir el estado del sistema y realizar las acciones necesarias para sincronizar su estado interno con el del sistema.
- *Remoción de componentes:* permite remover comportamiento innecesario, usualmente como resultado de recientes adiciones que suplantan el comportamiento original. Las condiciones para una remoción apropiada son gobernadas por la aplicación específica.
- *Reemplazo de componentes:* por lo regular se considera un caso especial de una adición seguida por una remoción cuando dos propiedades adicionales son requeridas: (1) el estado de ejecución debe ser transferido a un nuevo componente (2) ambos componentes no deben estar activos simultáneamente durante el cambio.
- *Reconfiguración:* básicamente consiste en realizar una reconfiguración estructural que soporte la modificación de las funcionalidades existentes para modificar todo el comportamiento del sistema.
- *Reestructurar o cambiar la arquitectura:* cambiar la organización o la arquitectura del sistema, desde cambio en componentes, conectores, incluso cambios de orden mayor como patrones de diseño o estilos arquitectónicos.
- *Aprovisionamiento de recursos:* consiste en la utilización de recursos adicionales en diferentes niveles, puede considerar una *extensión al reemplazar, adicionar y remover* cualquier recurso, por ejemplo, un nuevo servidor.

- *Reiniciar y desplegar de nuevo:* se reinician algunas o todas las entidades del sistema a diferentes niveles principalmente debido a fallos o defectos.

4.3 OTRAS TÉCNICAS DE ADAPTACIÓN

La sección precedente se enfocó en técnicas genéricas que soportan parte de los marcos de trabajo para la adaptación, la dificultad de hacer cambios, incluso usando estas técnicas, varía enormemente de una aplicación a otra, todo depende del tipo de cambio requerido y de la naturaleza de la arquitectura del sistema. También algunos estilos arquitectónicos y arquitecturas están habilitados para manejar más ciertos tipos particulares de cambios que otros. A continuación, se describen algunas técnicas más específicas de adaptación.

4.3.1 Aproximación orientada a interfaces

Se han categorizado como enfocada a interfaces ya que descansan sobre interfaces de programación de la aplicación (API's) o interfaces propias de la aplicación. Estas técnicas no soportan todos los tipos de cambio y su enfoque primario es solo adicionar funcionalidad en la forma de un nuevo módulo.

- *Interfaces de Programación de Aplicación(API).* En esta técnica la aplicación expone una interfaz, un conjunto de funciones, tipos y variables, para los desarrolladores. Los desarrolladores pueden crear y enlazar nuevos módulos que usan estos elementos de la interfaz en la forma en que los ellos lo elijan. El componente adicionado puede hacer uso de cualquiera de las características de la aplicación original que son expuestas por la API, pero no puede acceder a interfaces de cualquiera de los componentes internos para la aplicación, y no puede modificar ninguna de las conexiones internas para la aplicación. El componente adicionado hace llamadas a la aplicación original; nada en la aplicación original puede hacer llamadas al componente adicionado.

- *Plug-Ins.* Esta técnica es una *imagen-espejo* de las API's. En lugar de adicionar componentes invocando la aplicación original, la aplicación hace invocaciones al nuevo componente. Para lograr esto, la aplicación original predefine una interfaz que una tercera parte conectada debe implementar. La aplicación usa e invoca la interfaz del plugin, de esta manera altera su comportamiento.

- *Arquitectura Componente/Objeto*. La aplicación original expone sus entidades internas y sus interfaces a desarrollos de externos, así que estas entidades pueden ser usadas durante el proceso de adaptación. La aplicación deja de considerarse como una entidad monolítica cuya estructura interna es opaca e inmune. Aquí, los complementos alteran el comportamiento de la aplicación residente adicionando nuevos componentes que interactúan con componentes existentes.

- *Lenguajes de script.* La aplicación provee su propio lenguaje de programación (el "*lenguaje de scripting*") y en un entorno en tiempo de ejecución. El desarrollador usa este lenguaje para implementar complementos, que cuando son ejecutados por el intérprete, alteran el comportamiento de la aplicación. Los lenguajes de script comúnmente proveen construcciones del lenguaje específicas del dominio y funciones propias que facilitan la implementación de complementos.

- *Interfaces de evento.* En esta técnica la aplicación original expone dos interfaces distintas para desarrolladores externos a la aplicación. La primera es una interface de entrada de eventos, que específica los mensajes que pueden recibir y frente a los cuales actuar, la segunda, una interfaz de salida de eventos, que especifica los mensajes que el aplicativo genera. El *mecanismo de eventos* actúa como un intermediario, encapsulando y localizando decisiones de comunicación y enlaces de un programa. Como un resultado, esas decisiones pueden ser alteradas independientemente de la aplicación original o de los componentes complementarios.

4.3.2 Aproximación basada en arquitectura

Esta es una aproximación más enriquecida que la adaptación enfocada a interfaces, dado que ofrece elementos de mayor complejidad a la “simple” adición de componentes. El núcleo de este tipo de solución es un modelo de arquitectura explícito que debe ser fiel a la implementación, y que servirá de base para razonar sobre los cambios del sistema. Si se hace una utilización consistente de un estilo arquitectónico subyacente se puede eliminar algunas de las dificultades inherentes asociadas con la consistencia entre modelo e implementación, aquí el tema fundamental es el relacionado a los enlaces, si un estilo arquitectónico provee conectores explícitos, es decir, está habilitado para atar y desatar componentes entre ellos, dicho estilo facilita las acciones de adaptación.

4.3.3 Aproximación basada en modelos formales

Adaptación basada en *ADL’s* (*Lenguajes de descripción de arquitectura*) son notaciones formales para expresar y representar diseños y estilos arquitectónicos, permiten la modificación dinámica de la arquitectura software.

Con los años ha habido una proliferación de lenguajes para la especificación de configuraciones de diseños complejos. Por mencionar algunas de sus características:

- Estos permiten especificar un programa distribuido como una construcción jerárquica de componentes.
- Los componentes interactúan accediendo a servicios
- Los componentes son vistos en términos de los servicios que proveen y los servicios que requiere.
- La ubicación de los componentes es transparente para el resto, generando de esta manera independencia del sistema del cual forman parte, propiedad conocida como independencia del contexto.

El propósito de estos lenguajes de especificación es permitir a los arquitectos construir tipos de componentes básicos y compuestos, declarando instancias de componentes y enlaces entre los servicios requeridos y los proveídos por otros utilizando una notación formal.

4.3.4 Aproximación basada en middleware configurable

El diseño de *middleware configurable* está basado en el concepto de reflexión, la técnica de reflexión tiene por meta básica exponer la implementación subyacente, de esta manera es posible insertar comportamiento adicional para monitorear dicha implementación e incluso puede llegar a ser usado para adaptar el comportamiento interno del sistema.

El diseño de *middleware configurable* introduce una arquitectura independiente del lenguaje usando un metamodelo para estructurar un meta-espacio y un grafo de objetos para mantener la estructura de composición de la aplicación. El middleware emerge como un componente de la arquitectura importante para aplicaciones distribuidas, su rol es presentar un modelo unificado de programación para temas de heterogeneidad y distribución.

El principio rector es la estructura de meta-espacio como un número de modelos de meta-espacios distintos, pero cercanamente relacionados, el objetivo es mantener la separación de preocupaciones entre los aspectos de composición, encapsulación y entorno.

4.4 ADAPTACIÓN AUTÓNOMA ON-THE-FLY

Realizar adaptación mientras el sistema está en operación, conocida como adaptación *on-the-fly* presenta varios retos, esto puede llegar a complicar el proceso de adaptación. La adaptación autónoma significa que no hay un

humano involucrado en el proceso o ciclo de adaptación, o que hay una dependencia mínima del sistema hacia este.

Los temas adicionales para realizar la adaptación *on-the-fly* son los siguientes:

- *Estrategia a nivel de servicio:* consiste en que la funcionalidad del sistema durante la ejecución de una adaptación puede ser mantenida a través del uso de componentes auxiliares, o pueden verse negativamente impactada o degradadas a un nivel bajo de funcionalidad. Básicamente se enfoca en responder a las preguntas ¿continuará el sistema con un servicio temporal mientras se lleva a cabo el cambio?, ¿continuará con un nivel de servicio reducido? ¿se perderá el servicio por completo?

- *Sincronización:* se refiere a qué condiciones mínimas debe cumplir el sistema para determinar el momento en que puede procesarse la adaptación. Por ejemplo, la sustitución de un componente puede solo ser permitido cuando ha finalizado de servir las peticiones de este, y no inicia ninguna petición sobre ningún otro componente de la aplicación. El principio es que es seguro realizar tal cambio sólo cuando no afecte adversamente la funcionalidad del resto de la aplicación.

- *Transferencia de estado:* ¿Si un componente es reemplazado, el componente sustituido necesita conocer alguna o toda la información de estado mantenida por su predecesor? Cuando un componente A es reemplazado con un componente B, la pregunta a resolver es ¿cómo el estado del componente B debería inicializarse para su puesta en marcha? La situación más fácil, es una donde la transferencia o recreación del correspondiente estado no es necesario. Una estrategia efectiva para conducir la transferencia de estado es simplemente externalizar, es decir, se copia el estado necesitado a un tercero, cuando el componente reemplazado se encuentre en línea su primera acción será inicializarse a sí mismo dirigido por el tercero que tiene la copia del estado anterior. Esta

estrategia, por supuesto, asume que los componentes fueron diseñados para sustitución, una condición no con frecuencia verdadera.

- Condición de inactividad: la inactividad consiste en que el componente no reciba ni envíe información, e internamente esté ocioso (es decir, todos sus hilos estén ociosos). Esto es, un componente C_i puede iniciar un conjunto de acciones que son realizadas por otros componentes, estableciéndose a sí mismo "inactivo" hasta que los otros componentes completen sus tareas y retornen el control a C_i. En este caso, C_i no está inactivo hasta que todas las acciones que son parte de la transacción estén completas.

Mientras los principios y temas discutidos por lo regular aplican de forma general a arquitecturas, hay algunas prácticas y aproximaciones que definitivamente facilitan el control de ciertos problemas prácticos para soportar adaptación *on-the-fly.* La práctica principal (o el principio más importante) es usar conectores explícitos, y usar componentes sin estado.

Un ejemplo exitoso se puede observar al considerar la arquitectura de la *Web*. La Web se considera basada en estos principios, y se encuentra continuamente en cambio. *Proxies* y *caches* pueden ser adicionados y borrados; nuevas tecnologías de *servidor web* localizados en diferentes nodos, nuevos *navegadores* instalados, además funciona debido a que las interacciones entre componentes son a través de mensajes discretos donde el protocolo núcleo (*HTTP*) requiere interacciones libres del contexto (*sin estado*) es decir el componente servidor no necesita mantener un historial de interacciones con el cliente para saber cómo responder una próxima petición.

El cambio autónomo en la adaptación subyacente consiste en efectuar un proceso de cambio sin intervención humana. El deseo de evitar intervención humana obedece a que *un sistema auto-adaptativo puede cambiar más velozmente que uno que involucre personas en el ciclo de adaptación,*

consecuentemente, un cambio auto-adaptativo es menos costoso. Las dificultades que conlleva un cambio autónomo involucran todas las actividades discutidas previamente, la observación de datos, el análisis y reconocimiento de cambios, la planeación y el despliegue las modificaciones del sistema en tiempo de ejecución, y más aún, todos los procesos ejecutables y toda la información involucrada en su ejecución, debe residir en el sistema autónomo.

4.5 ADAPTACIÓN BASADA EN SISTEMAS MULTIAGENTE

La arquitectura basada en agentes es un estilo arquitectónico específico que impone restricciones particulares sobre un sistema. Esta sección tratará trabajos relacionados con la adaptación basada en sistema multi-agente, con el fin de aportar un panorama que permita reconocer el aporte del siguiente capítulo donde se define la propuesta de marco de trabajo.

Según K. Geihs[63], los sistemas adaptativos pueden tener diferentes facetas divididas en dos grandes grupos: de tiempo de diseño y de tiempo de ejecución, siendo el último el más complejo. De lo anterior aparecen nuevos paradigmas que buscan la creación de sistemas capaces de adaptar su comportamiento automáticamente estudiados desde las áreas de computación como la computación autonómica[64], como ya se ha discutido, o también desde la computación orgánica[65] que soportan la búsqueda de soluciones a los retos que estos sistemas plantean. Como exponen Salehie y Tahvildari, las aproximaciones que introducen conceptos de control centralizado son llamados sistemas auto-adaptativos, en contraste con las arquitecturas

[63] K. Geihs, “Selbst-adaptive software,” Informatik-Spektrum, vol. 31, 2008, pp. 133–145

[64] J. O. Kephart and D. M. Chess, “The vision of autonomic computing,” Computer, vol. 36, no. 1, 2003, pp. 41–50.

[65] J. Branke, M. Mnif, C. Muller-Schloer, H. Prothmann, U. Richter, F. Rochner, and H. Schmeck, “Organic computing - addressing complexity by controlled self-organization,” in Proc. of the 2th Int. Symp. on Leveraging Appl. of Formal Methods, Verification and Validation, ser. ISOLA ’06. IEEE Comp. Soc., 2006, pp. 185–191.

descentralizadas que utilizan ciclos de retroalimentación distribuidos y mecanismos de coordinación conocidos como sistemas auto-organizados.

Basado en lo anterior, en Preisler[66] se propone una aproximación consistente en una arquitectura de sistema genérica que gobierna el desarrollo de sistemas auto-organizados basados en sistemas MAS.[67] , también aporta un lenguaje de alto nivel que facilita la descripción declarativa de procesos de coordinación, prescribiendo adaptaciones estructurales que pueden ser interpretados por un marco de trabajo y complementado con un proceso de ingeniería que consiste en actividades de desarrollo incrementales que conducen la manifestación de comportamiento auto-organizado.

El núcleo del proceso de adaptación se basa en un sensor que se ubica en cada agente, el cual detecta cualquier decremento de los indicadores de rendimiento del sistema, esto inicia un proceso de acuerdo distribuido (entre agentes) que permite el intercambio o reconfiguración, así el sistema puede adaptarse a las condiciones cambiantes de manera automática.

Otros trabajos en esta línea de estudio se han basado en la descentralización por considerarla crucial y potenciadora de requerimientos de calidad tales como apertura, robustez y escalabilidad, tal es el caso de Weyns y Georgeff[68] quienes exponen que un sistema multi-agente pertenece a una clase de sistemas descentralizados en la que cada componente (agente) resuelve problemas de forma autónoma al estar dotado de capacidades para operar en entornos dinámicos y de incertidumbre. La interacción entre ellos los habilita para resolver problemas, más allá de sus capacidades y conocimiento individual. Además, exhiben características que motivan su uso en la

[66] Preisler, T., & Renz, W. Structural Adaptations for Self-Organizing Multi-Agent Systems.
[67] Acronimo para Multi-Agent Systems. Mejor explicados en. M. Woolridge. Multi-Agent Systems: An Introduction. John Wiley & Sons (Chichester, England), 2001.
[68] Danny Weyns and Michael Georgeff. Self-Adaptation Using Multiagent Systems. Software technology. IEEE Software 2010.

construcción de sistemas autoadaptables tales como: bajo acoplamiento, sensibilidad al contexto[69], resistencia a falla y a eventos inesperados.

Los agentes se han utilizado para realizar procesos de adaptación en diversos estilos arquitectónicos, tal es el caso de sistema basado en agentes para una arquitectura orientada a servicios de auto-adaptación[70]. En este trabajo los servicios son supervisados por agentes autónomos los cuales son responsables por decidir cuáles servicios deberían ser elegidos para interoperación. Los agentes aprenden a elegir estratégicas autónomamente utilizando *aprendizaje supervisado*. Este tipo de aprendizaje mejora la experiencia[71] frente a las estrategias simples tales como selección aleatoria y estudios relacionados han mostrado que es más eficiente[72].

El aprendizaje supervisado puede ser utilizado por agentes para generar estrategias para adaptación de arquitectura orientada a servicios. Debido a la representación simbólica, el conocimiento aprendido puede ser analizado y editado por expertos humanos, esto permite identificar patrones no conocidos a priori que puede ser usado por expertos para entender el comportamiento de sistemas distribuidos y mejorar su configuración. En resumen, la idea fundamental del sistema es que un servicio pueda estar acompañado por un agente, el cual es responsable por elegir otros servicios para procesar solicitudes dadas. El agente está aprendiendo como utilizar servicios utilizando aprendizaje supervisado. Así el sistema consiste en un conjunto S de servicios, cada servicio s(i) puede inter-operar con otro servicio usando acciones publicadas por estos. Una acción puede ser implementada por varios servicios

[69] Vokovik investiga aplicaciones sensibles al contexto, que están habilitadas para adaptar a cambios en el entorno en su trabajo. Context Aware Service Composition. PhD thesis, University of Cambridge, 2006.

[70] Bartłomiej Śnieżyński. Agent-based Adaptation System for Service-Oriented Architectures Using Supervised Learning. Procedia Computer Science. Volume 29, 2014, Pages 1057–1067

[71] Bartłomiej Śnieżyński and Jacek Dajda. Comparison of strategy learning methods in farmerpest problem for various complexity environments without delays. Journal of Computational Science,
4(3):144 – 151, 2013.

[72] Bartłomiej Śnieżyński. Agent strategy generation by rule induction. Computing and Informatics, 32(5):1055–1078, 2013.

s(i1), s(i2), ..., s(ik) puede usarse un método de aprendizaje para construir una estrategia basada en condiciones actuales del sistema y del entorno.

Otro ámbito de empleo de agentes está en el middleware[73], donde es concebido el mismo es definido como una red de agentes, con el objetivo de superar el reto de adaptación en tiempo de ejecución en un sistema abierto. El principal aporte de este trabajo es encontrar una sinergia entre los métodos de ingeniería de software y la tecnología de agentes, para modelar y desarrollar sistemas adaptativos. El diseño propuesto para este middleware se basa en 3 capas, la capa inferior provee servicios de infraestructura a las capas superiores. La capa intermedia sirve de envoltura que encapsula los componentes de información de bajo nivel. La capa superior se compone de *agentes especializados* que manejan acciones del ciclo de adaptación coordinándose con otros agentes.

Los agentes especializados destinados a las tareas de adaptación son: agentes de monitoreo del contexto, quienes introducen la característica de sensibilidad al contexto al middleware. Los agentes de selección de configuración, que tienen por objetivo planear y evaluar, estos calculan la utilidad de seleccionar configuraciones para la aplicación mediante la asignación de pesos ayudando a razonar sobre las alternativas de manera determinista. Los agentes de ejecución su principal meta incluye la continuidad del servicio mientras un script de adaptación produce los cambios. Finalmente, el agente visualizador despliega información para el usuario sobre el proceso de adaptación y notificaciones mientras el middleware realiza su trabajo.

Otros trabajos han mostrado como una arquitectura jerárquica basada en agentes potencia la escalabilidad en sistemas auto-organizados y autoadaptables. Las particiones funcionales entre niveles de agentes de

[73] Qureshi, N. A., & Perini, A. (2008, August). An agent-based middleware for adaptive systems. In 2008 The Eighth International Conference on Quality Software (pp. 423-428). IEEE.

monitoreo y reconfiguración no introducen cuellos de botella en un nivel en particular[74]. Este tipo de arquitectura ha permitido introducir elementos que permiten adoptar diferentes métodos de optimización para solución de problemas en sistemas con alta complejidad. Entre estos métodos se encuentra algoritmos de optimización evolutiva para SoS[75] expuestos en Minghong, 2014 [76]. Tal diseño permite a los agentes ganar una mayor adaptabilidad basada no solo en el impacto de su comportamiento local acorde con su estado y experiencia, sino también en el intercambio de la experiencia de otros agentes en el MAS para inferir restricciones del entorno, que pueden contribuir positivamente al alcance de sus propias metas[77].

Esta sección ha citado algunos trabajos relacionados con los sistemas complejos adaptativos mediante el diseño de sistemas multi-agente (MAS), el siguiente capítulo se centrará en la especificación del marco de trabajo basado en los principios de los sistemas complejos adaptativos utilizando para su diseño la arquitectura de sistemas multi-agente presentando con claridad las razones que condujeron a esta elección.

4.6 CONCLUSIONES

- Son diversas las aproximaciones a sistemas adaptativos y este capítulo ha presentado una introducción a algunas de ellas, también a los diferentes mecanismos que realizan tareas específicas dentro del proceso

[74] GUANG, Liang; PLOSILA, Juha; TENHUNEN, Hannu. From self-aware building blocks to self-organizing systems with hierarchical agent-based adaptation. En *Proceedings of the 2014 International Conference on Hardware/Software Codesign and System Synthesis*. ACM, 2014. p. 23.
[75] Acronimo para referirse a Systems of Systems.
[76] PAN, Jiali; HAN, Minghong. The evolutionary optimization of system of systems based on Agent model. En *2014 IEEE International Conference on System Science and Engineering (ICSSE)*. IEEE, 2014. p. 231-235.
[77] JIAO, Wenpin; SUN, Yanchun. Self-adaptation of multi-agent systems in dynamic environments based on experience exchanges. *Journal of Systems and Software*, 2016, vol. 122, p. 165-179.

de adaptación con lo que se espera cubrir la mayoría de las dimensiones de estos sistemas.

- La parte final del capítulo realiza un resumen de trabajos relacionados al ámbito de la propuesta, ellos cubren diversas dimensiones de los sistemas multi-agente y sus instancias de aplicación en el desarrollo de software para sistemas que exhiben adaptabilidad.

5. PROPUESTA DE MARCO DE TRABAJO

El objetivo de este capítulo es presentar una propuesta de marco de trabajo para gobernar el proceso completo de adaptación, utilizando como marco de referencia los principios de la teoría de sistemas complejos para estructurar una aplicación autónoma. El propósito de un marco de trabajo es proveer al ingeniero una lista de chequeo de cuestiones para tener en cuenta a la hora de implementar un producto bajo una metodología, paradigma o patrón determinado y de esta manera explotar las ventajas o beneficios que este le puede aportar. No todas las partes del marco de trabajo serán igualmente útiles en todos los contextos de adaptación, pero se hace un intento serio de presentar los temas de forma cohesiva y coherente.

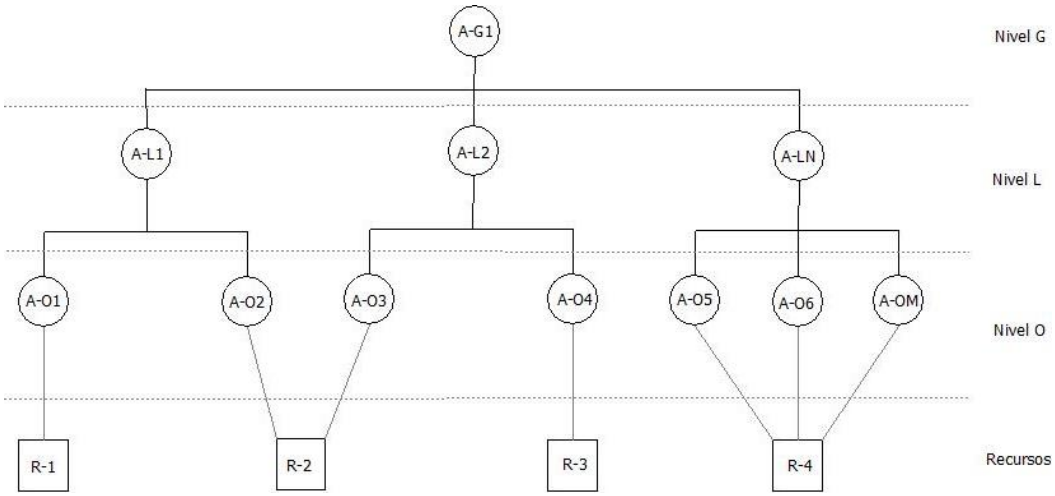
La primera aproximación a la definición del marco de trabajo propuesto reza de la forma:

> *Es un marco de trabajo adaptativo centrado en agentes heterogéneos, agrupados en una estructura jerárquica, mantenida por un modelo de propagación basado en el patrón visitante, y desplegada sobre los recursos del sistema*.

Hay varias palabras clave que señalan características que saltan a la vista en esta definición de marco de trabajo; centrado *en agentes heterogéneos* hace referencia a que todo elemento constitutivo es un agente, pero no todos los agentes son ni trabajan de la misma forma, esto es, cada agente pertenece a un *tipo de agente* diferenciado por su función específica. *Agrupados en una estructura jerárquica* hace referencia a la forma en que se organizan estos agentes, las estructuras jerárquicas tienen muchas características y ventajas propias de la topología que serán descritas posteriormente en detalle, una de

ellas es permitir realizar una distribución en niveles de los nodos de la jerarquía, de esta manera es posible realizar *separación de responsabilidades*, *división del trabajo*, *fragmentación de metas y controlar la granularidad en todos estos conceptos*. Por otro lado, el *modelo de propagación* mencionado se refiere a la manera en la que se llevarán a cabo recorridos o navegación sobre la estructura con el ánimo de realizar mantenimiento y control de esta. Finalmente, cuando se afirma que está *desplegada sobre los recursos del sistema* hace referencia a que cada agente en esta estructura se mapea con elementos tanto internos como externos del sistema.

Figura 13. Modelo de arquitectura centrada en agentes



Fuente. Elaboración personal

La imagen en la Figura 13 enseña una aproximación visual del modelo propuesto, este se refina a lo largo del capítulo agregando mayores detalles, sin embargo, este es un buen punto de partida, a continuación, se mencionan algunos detalles al respecto.

La imagen presenta tres niveles de agentes con tareas específicas lo cual implica un diseño diferente para cada tipo de agente, la estructura en la cual se organizan permitiría crear relaciones especializadas entre ellos como maestro-esclavo o padre-hijo incluso algunos elementos como se presenta en la

imagen pueden desplegarse sobre recursos que otros también poseen, obviamente sin convertirlo en un grafo. También, es posible explotar esta jerarquía con un diseño de modelo de propagación apropiado que facilite tareas de mantenimiento del sistema adaptativo como la creación y eliminación de agentes, sirve además de base para soportar fenómenos como la *coordinación* y la *emergencia*. Cada una de estas características será abordada con mayor detalle a lo largo del presente capítulo, el objetivo principal es definir claramente el modelo de marco de trabajo adaptativo propuesto y sus elementos constituyentes.

5.1 ARQUITECTURA DEL SISTEMA ADAPTATIVO

Esta sección tiene por objetivo introducir los conceptos de arquitectura del marco de adaptación propuesto.

5.1.1 Componentes

Los elementos constituyentes del sistema son los agentes, no está de más recordar su definición:

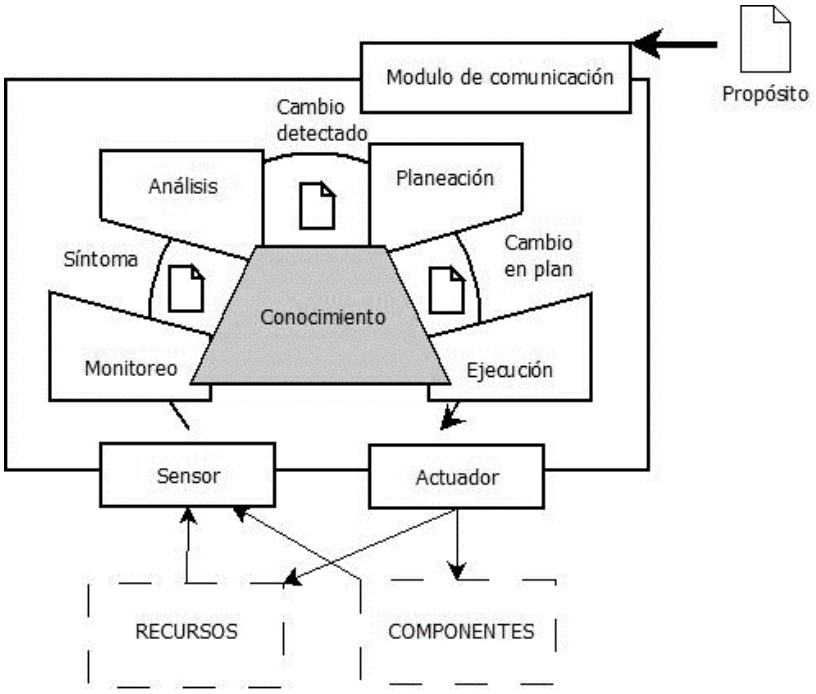
> *Es una entidad auto-adaptativa, la cual tiene percepción, adaptación y reacción a cambios del entorno, opera en entornos dinámicos. Con propiedades de autonomía, social y adaptabilidad.*

Lo anterior significa que un agente es una entidad capaz de percibir su entorno y de dichas observaciones realizar respuestas tendiendo a maximizar su funcionamiento en su entorno operativo. La propiedad de autonomía hace referencia a que el agente es “independiente”, es decir, puede trabajar por sí mismo sin intervención externa. La propiedad de ser social se refiere a que el agente debe tener cierto grado de comunicación con entidades externas.

Finalmente, la propiedad de adaptabilidad se refiere a que el agente puede responder rápidamente a cambios en el entorno.

La anterior descripción da unos puntos claves para tener en cuenta en el diseño de un agente y de alguna manera es posible relacionarla con el modelo de elemento autonómico descrito en un capítulo precedente la imagen de la Figura 14 recordará a que se refiere.

Figura 14. Modelo de elemento autonómico

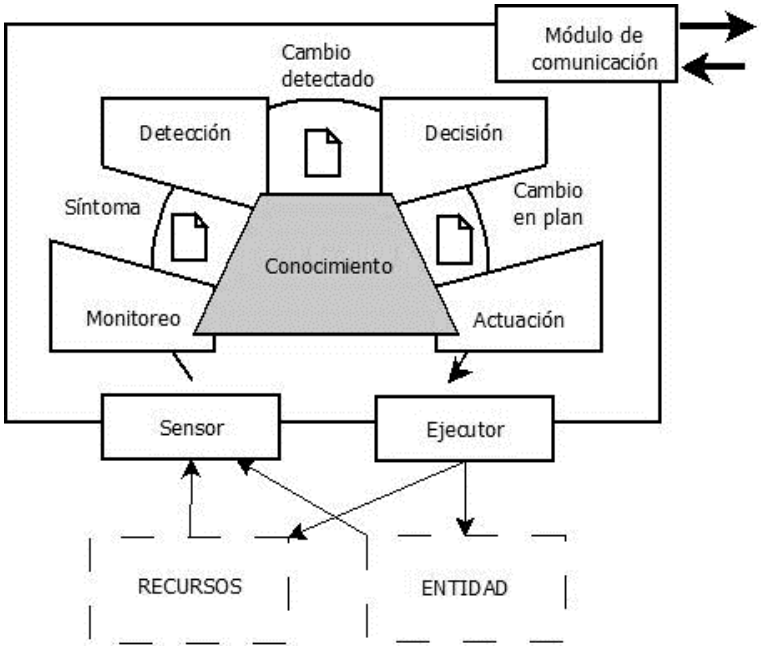


Fuente. Elaboración personal

Este modelo se ajusta muy bien a la definición presentada de agente e incluso satisface las propiedades previamente descritas, sin embargo, se manejan unos nombres ligeramente relacionados a las actividades que lo componen para tener una separación de responsabilidades más ajustada a dichas etiquetas, así, observando se ajusta el modelo como se muestra en la imagen de la Figura 15.

Cada agente parte con un conocimiento inicial que recoge información relacionada a su tarea específica, cada agente tiene un módulo sensorial que puede conectarse a una o varias variables bien sea del entorno o de la entidad constituyente de la aplicación, esto generará un flujo de datos del cual el agente intentará extraer regularidades que permitan refinar el esquema inicialmente albergado, es decir, datos previos del comportamiento del elemento y sus efectos, y crear planes de cambio que los ejecutores del cambio realizarán sobre el elemento que corresponda.

Figura 15. Modelo de agente con ajuste de etiquetas

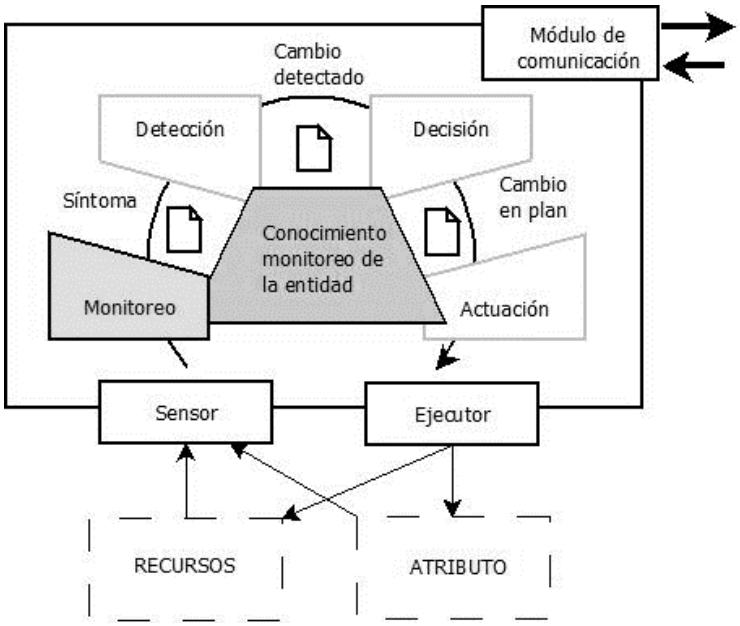


Fuente. Elaboración personal

La manera en que se desea utilizar en este marco de trabajo involucra la especialización de sus componentes, esto es, el modelo de elemento autonómico o agente enfatiza alguna de sus actividades en el ciclo de trabajo, esta es una recomendación en este marco de trabajo, agentes especializados en una tarea, el diseño de cada tipo de agente se describe en secciones posteriores en este capítulo, por ejemplo, un agente de monitoreo lucirá entonces como se presenta en la Figura 14. Lo anterior no quiere decir que el

agente pierda alguna de las propiedades ya mencionadas y que lo hacen un *agente*, por el contrario, esas actividades se seguirán realizando, pero para apoyar la especialidad del tipo de agente, el sentido de esto es que el *conocimiento adquirido* toma un carácter diferente que resaltará en el momento de la interacción con otros agentes.

Figura 16. Modelo de agente especializado



Fuente. Elaboración personal

5.1.2 Conectores

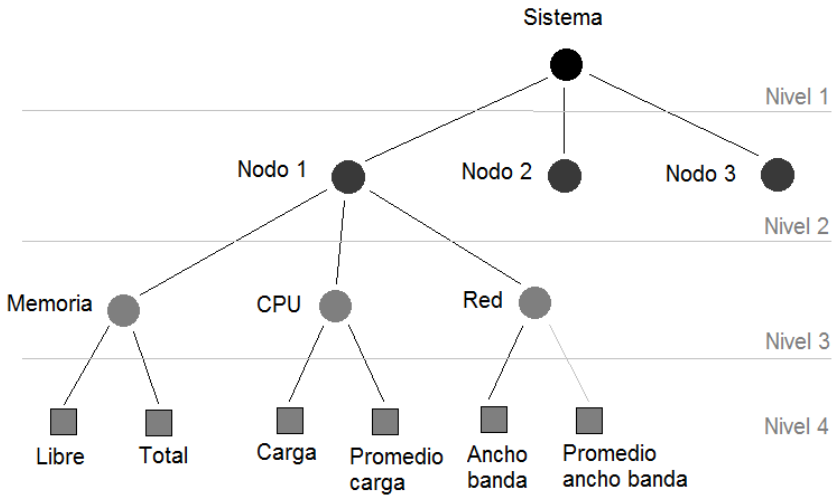
A este nivel se está más interesado en describir los mecanismos de interacción proporcionados por el marco de trabajo, que en hablar de tecnología específica. El modelo de agente mostrado tiene intercambio con el entorno en el que opera, los puntos donde se realiza este intercambio incluyen el módulo de sensores, el módulo ejecutor y el módulo de comunicación.

Los sensores permiten definir modelos de monitoreo en varios contextos. Un contexto debe entenderse como un dominio independiente y representar diferentes aspectos del entorno de ejecución y del estado del programa

(recursos, estado interno). Cada contexto está definido por un nodo n-ario que puede contener dos tipos de elementos: *atributos*, su principal función es obtener valores, y *recursos*, son colecciones de atributos, similares a carpetas en un sistema de archivos.

Un ejemplo de lo anterior se ilustra en la siguiente imagen con una jerarquía de ejemplo, un caso particular de contexto:

Figura 17. Jerarquía de sensores



Fuente. Elaboración personal

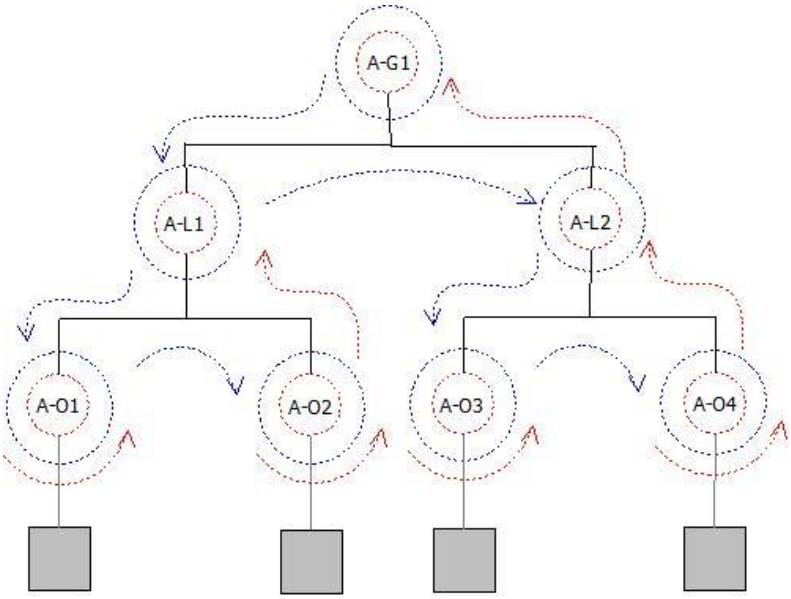
El sistema (recurso raíz) contiene varios nodos desplegados en el entorno (segundo nivel), los recursos de cada nodo (memoria, CPU, y red) se presentan como sub-recurso en el tercer nivel, finalmente cada recurso tiene algunos atributos describiendo su estado actual. Obviamente, un recurso puede representar un elemento constituyente del propio aplicativo, por ejemplo, una cola de mensajes, un nodo cliente, un distribuidor de carga, un módulo, etc.

Por otro lado el módulo ejecutor permite interactuar sobre el recurso o sobre el atributo, lo que pretende este módulo es establecer parámetros del elemento

para de esta forma provocar la adaptación, aquí se puede tener un mecanismo similar al de los sensores, pero donde el establecimiento del valor de un nuevo atributo desemboque en un conjunto de acciones de cambio para este, en caso de ser, del cambio de un recurso se deben establecer los lineamientos para afectar los valores de cada atributo en este.

¿Ahora bien qué pasa con el módulo de comunicación, y cuál es su utilidad en este marco de trabajo? Para descifrar este punto hay que recordar que la propuesta sugiere una organización jerárquica de los agentes, las razones para este tipo de organización se dan en la siguiente sección, pero es necesario mencionarlo aquí porque este tipo de topología condiciona la interacción entre los agentes.

Figura 18. Modelo de propagación



Fuente. Elaboración personal

La imagen de la Figura 18 deja explícito las formas en que se pueden realizar un recorrido al árbol, estas se resumen a continuación:

Pre-Orden: para recorrer un árbol no vacío en pre-orden, hay que realizar las siguientes operaciones visitar la raíz del árbol, recorrer el subárbol izquierdo y recorre el subárbol derecho.

In-orden: para recorrer un árbol no vacío en in-orden, hay que realizar las siguientes operaciones recorrer el subárbol izquierdo, visitar la raíz y recorrer el subárbol derecho.

Post-orden: para recorrer un árbol no vacío en post-orden, hay que realizar las siguientes operaciones recorrer el subárbol derecho, recorrer el subárbol izquierdo y visitar la raíz.

Dado lo anterior y combinado con un patrón de diseño como el *visitante* es posible especializar las interacciones entre los agentes. El objetivo de dicho patrón es separar el algoritmo de la estructura del objeto, la idea básica es recorrer la jerarquía de agentes de diferentes heterogéneos utilizando alguno de los recorridos referidos y recogiendo información bien sea: que los nodos padres quieran comunicar a los nodos hijos, o viceversa. Implica que es posible diseñar diferentes tipos de comunicaciones entre los agentes y aplicarlas utilizando un hilo visitante diferente para cada uno.

Por poner un ejemplo de lo anterior, suponga que se establece una nueva meta en un nodo de la parte superior de la jerarquía, pueden haber diferentes mecanismos para comunicar ésta a todos los nodos, una forma sería tener un *blackboard* que los agentes podrían consultar, otra podría conectar todos los agentes con todos los demás, la propuesta de este proyecto es crear un visitante especializado en transferir esta información a cada agente, dependiendo de su posición la jerarquía puede tomar la decisión de cambiar el contenido del visitante con el ánimo que cuando este vaya a los nodos inferiores las metas estén más acorde al papel que juega dicha ramificación en la jerarquía.

5.1.3 Topología

Como ya se ha mencionado recurrentemente los agentes se organizan en una estructura jerárquica o árbol n-ario. Estos cuentan con varias características que se analizan en esta sección y permite utilizar la metáfora de sistema de sistemas para involucrar ciertos conceptos clave del marco de trabajo propuesto.

Una definición formal de árbol se da a continuación:

> *Un árbol es un grafo con dos características, es conexo y no tiene ciclos*.

Un grafo o gráfica G es un conjunto *V* de vértices o nodos, y un conjunto de *E* de aristas tal que cada artista *e* ϵ *E* queda asociada a un par no ordenado de vértices. *En teoría de grafos, un grafo* G se dice conexo si, para cualquier par de vértices *u* y *v* en *G,* existe al menos una trayectoria (una sucesión de vértices adyacentes que no repita vértices) *de u* a *v.* La característica de no tener ciclos significa que para cualquier trayectoria de longitud mayor a cero entre cualquier par de nodos *v, u* nunca se da el caso en que *v* = *u*.

Una referencia importante para selección de este tipo de organización está basada en la definición de propuesta por Mo Jamshidi sobre sistemas de sistemas (SoS en adelante):

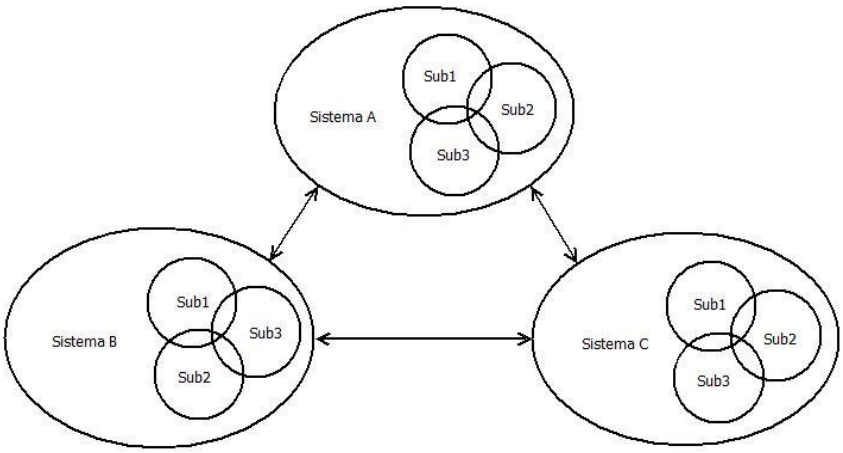
> *"Un sistema de sistema es un número finito de sistemas constituyentes los cuales son independientes y operativos, y que constituyen una red de trabajo por un periodo de tiempo para alcanzar un objetivo de alto nivel".*

Una de las razones para pensar en este modelo como base para el desarrollo del marco de trabajo propuesto son las cinco características que *Sage* y

Cuppan. Las cinco características más importantes de los sistemas de sistemas son la operatividad, la independencia, distribución geográfica, comportamiento emergente, y desarrollo evolutivo. Sin embargo, ellos se enfocan en el aspecto evolutivo del sistema complejo adaptativo. Mientras esta propuesta busca centrarse en el aspecto adaptativo, una diferencia que se explicara con mayor detalle en una sección posterior.

Para entender un poco la dinámica de los SoS la imagen de la Figura 19 da una mejor apreciación del sentido de la relación con el tema en discusión.

Figura 19. Sistema de sistemas



Fuente. Elaboración personal

Como se aprecia los sistemas A, B y C son operativos e independientes, pero los subsistemas {1, 2, 3} no son independientes. La adaptación del sistema dado el modelo de agentes, donde estos operan en un entorno dinámico y tienen la percepción y reacción a los cambios del entorno, está definida por los cambios en la estructura, estado, comportamiento y función en el tiempo de los sistemas constitutivos.

Dada la organización jerárquica propuesta es posible explotar ventajas como las siguientes:

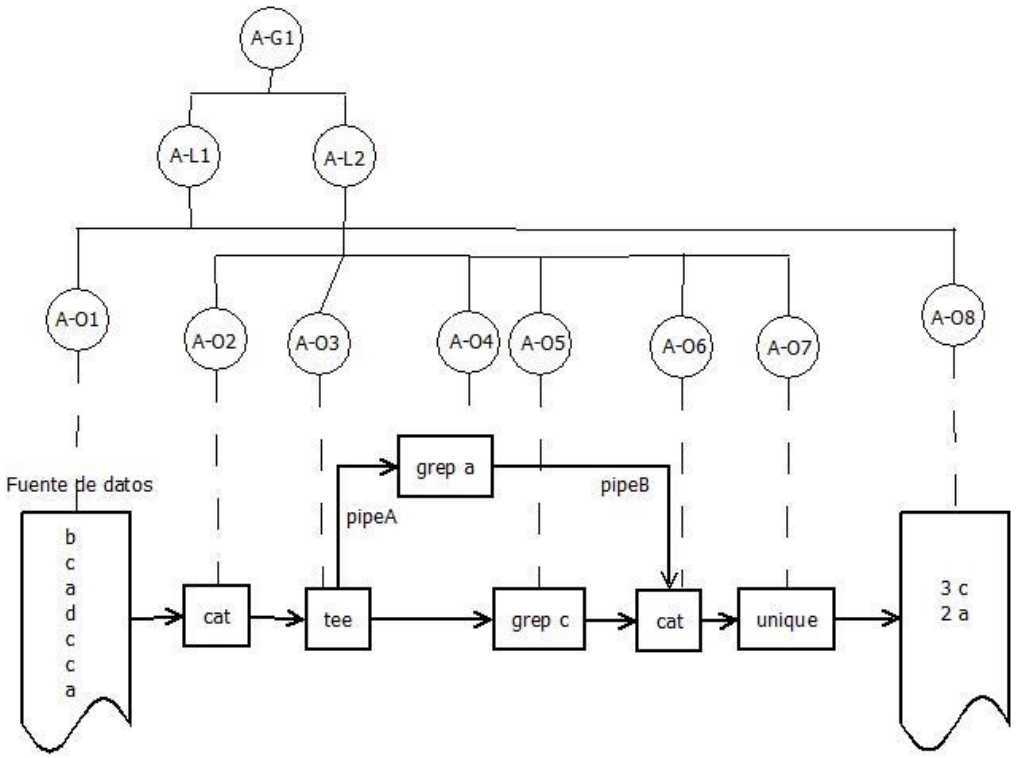
Subárboles: se puede especializar ramificaciones enteras con agentes específicos para ciertas tareas, por ejemplo, agentes encargados de fragmentar metas o subdividir tareas.

Niveles: cada nivel puede dar la flexibilidad suficiente para variar la granularidad de las actividades llevadas a cabo por un agente. Por ejemplo, realizar ajustes según el grado de resolución asociado al nivel del agente, así un agente podría realizar ajustes a nivel de una estructura de datos específica (bajo nivel) mientras otro en un nivel superior podría sugerir la creación de un nuevo componente.

Modelo de propagación: como se ha mencionado repetidamente permite establecer interacciones, bien sea de los nodos de niveles superiores hacia los nodos niveles inferiores (a modo de intervención) o viceversa (a modo de emergencia).

Descomposición: este tipo de estructura se ajusta muy bien a las necesidades requeridas de descomposición de cualquier sistema sin importar el tipo de estilo arquitectónico del sistema analizado, por dar un ejemplo, si se elige un estilo arquitectónico como Pipes/Filters, es posible crear una estructura jerárquica superpuesta al estilo arquitectónico elegido, la imagen de la Figura 20. ilustra este punto.

Figura 20. Descomposición del sistema



Fuente. Elaboración personal

Es importante anotar que la anterior no es la única estructura jerárquica que puede ser construida para esta aplicación específica del estilo pipes/filters, dicha estructura depende del análisis del ingeniero de la aplicación.

Separación de responsabilidades: uno de los aspectos de mayor discusión en la construcción de este tipo de sistemas es el relacionado con la posición del sistema adaptativo con respecto al sistema objeto. No es posible, en esencia tener una clara definición de dónde termina el sistema objeto y donde comienza el sistema adaptativo, en breve se discute esta afirmación. Sin embargo, como sugiere la imagen de la Figura 20 el marco de trabajo propuesto trata de desplegarse sobre los elementos constitutivos del sistema, dejando la mayor parte del trabajo de adaptación a los agentes componentes del árbol, separando por lo menos en teoría el sistema observado del sistema adaptativo, tal como lo sugiere en su libro Murray Gell-Mann. El quark y el jaguar, aventuras en lo simple y lo complejo.

En las secciones posteriores se estudia en detalle cada etapa del ciclo de adaptación, finalmente, se presenta un conjunto de conceptos transversales al marco de trabajo como son la emergencia, atributos de calidad y la optimización.

5.2 COMPONENTE DE CAPTURA DE DATOS

La parte baja de esta jerarquía muestra un conjunto de elementos que se ha llamado recurso, este nombre no hace referencia exclusiva a un elemento interno del sistema, cómo un objeto, un componente, una estructura de datos, sino también a cualquier elemento externo que nos pueda brindar información del entorno en el cual funciona el aplicativo, por ejemplo, la memoria RAM, la CPU, el disco duro, etc.

De alguna manera si se tiene acceso a estos elementos, se puede acceder a información de su estado o comportamiento en el tiempo. Por poner un ejemplo, imagine una cola de mensajes un elemento común en un estilo arquitectónico como un Message Broker Architecture o MBA por sus siglas en inglés. Considere un agente de tipo observador sobre este elemento, inmediatamente se sugieren preguntas como: ¿Cuál es el estado de la cola, es decir, tiene mensajes? ¿Qué capacidad tiene? ¿Aún tiene disponibilidad?, o se pueden sugerir preguntas más relacionadas a su comportamiento en el tiempo ¿Cuál ha sido la frecuencia de mensajes de entrada para esta cola? ¿Cuál ha sido la frecuencia de salida de mensajes de esta cola? ¿Alguna vez se ha desbordado esta cola? Estos son solo algunos ejemplos de información o datos que el agente observador puede estar interesado en consultar sobre este recurso.

Los recursos serán definidos de la forma:

Son contenedores de colecciones de atributos, los atributos pueden ser estáticos o dinámicos, y pueden obtenerse de la lectura directa de propiedades o mediante cálculos elaborados.

El diseño del componente de acopio de datos en este marco de trabajo debe tener las siguientes consideraciones:

- Considerar la *resolución*, este es un concepto físico que se refiere al detalle con el cual deberían hacerse las observaciones, para definir una forma de complejidad siempre es necesario acotar el grado del detalle en la descripción del sistema, en este caso, es diferente analizar el comportamiento de la cola de mensajes mencionada, contra el flujo de cada mensaje en ella, la información obtenida en ambas resoluciones con seguridad será exactamente.
- Considerar la *heterogeneidad*, es necesario tener presente que los elementos constitutivos de cualquier arquitectura tienen propiedades y comportamientos diferentes, lo cual implica que el marco de trabajo debería preocuparse por imponer un grado mínimo de estandarización en la descripción de estas características.
- Considerar la *longitud de la descripción*, este concepto está relacionada a la estructura del lenguaje empleado para realizar la descripción de las observaciones, es muy importante dadas las restricciones técnicas de almacenamiento de datos que la tecnología impone. Como ejemplo imagine un grafo constituido por 6 nodos, etiquetados con letras (A, B, C, D, E, F) y por un conjunto de aristas de cada nodo a todos los demás. Las siguientes dos descripciones mencionan exactamente lo mismo, pero su longitud es completamente distinta:

  Forma 1: {(A, B), (A, C), (A, D), (A, E), (A, F), (B, C), (B, D), (B, E), (B, F), (C, D), (C, E), (C, F), (D, E), (D, F), (E, F)}.

  Forma 2: Todos los nodos están unidos.

5.3 PROCESO DE MONITOREO

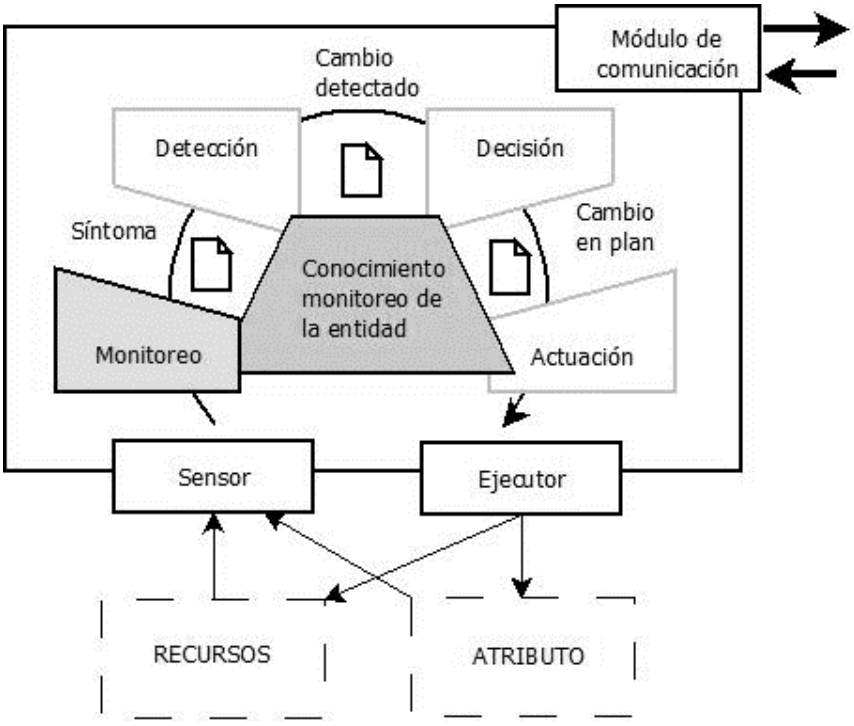
Ya se ha mencionado que el componente de captura de datos es un paquete de información sobre atributos del sistema, ahora se está más interesado en el mecanismo utilizado para emplear dicho componente, el objetivo de esta sección es diseñar el componente de monitoreo.

Como ya se ha planteado cada elemento dentro del marco de trabajo es un agente, cada agente se considera un elemento independiente que como se ha sugerido tiene una tarea especializada, la tarea que en este momento se centrará el estudio es en la actividad de monitoreo. Si bien los elementos de captura de datos recogen atributos en el tiempo bien sea rasos o semi-calculados, aún queda por definir algunos puntos de especial interés.

Para iniciar esta discusión se plantea la definición básica del proceso de monitoreo junto con su objeto:

> *Los monitores y sensores son componentes de que observan los recursos con alguna importancia para las propiedades self-* del sistema y cuyo principal objetivo es resumir en síntomas dichas observaciones.*

Figura 21. Agente de monitoreo



Fuente. Elaboración personal

Dada la definición se pueden distinguir dos tipos de monitores:

1. *Monitor pasivo*: hace referencia a los mecanismos subyacentes del entorno para realizar auto descripción. Un ejemplo de esto pueden ser los comandos de consola que arrojan información del entorno de ejecución, como ***ps*** en Linux que reporta una lista de los procesos actuales en el sistema.
2. *Monitor activo*: este hace referencia a modificar el software en algún nivel para obtener cierta información del entorno o de sistema objeto de evaluación no proporcionado por otros mecanismos.

Algunas consideraciones a la hora de llevar a cabo el diseño de este agente se presentan a continuación:

- Considerar la *frecuencia de monitoreo,* la frecuencia es el número de veces que aparece, sucede o se realiza la observación de atributos

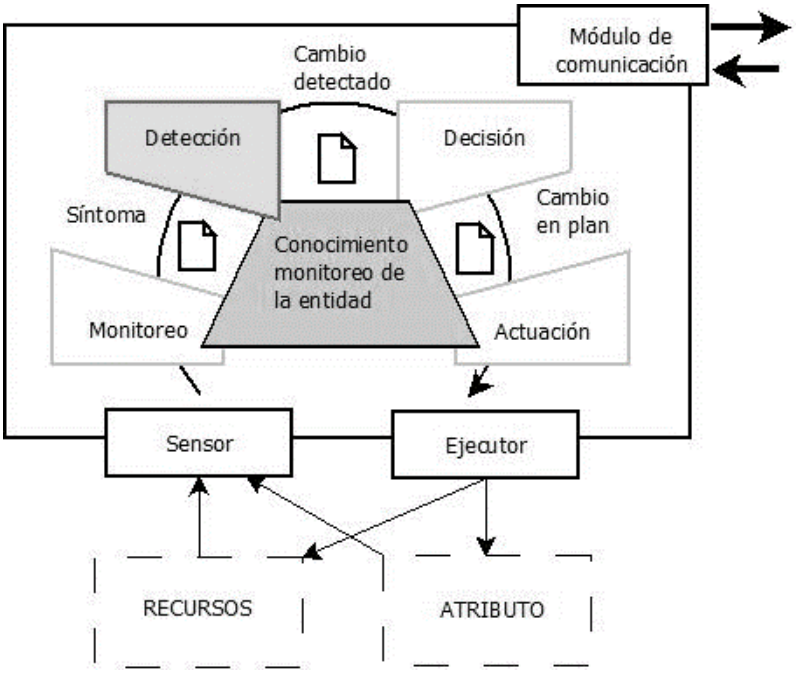
durante un período determinado, es importante que el diseñador elija una frecuencia apropiada, dado que si los atributos observados varían rápidamente puede darse el caso que sean imperceptibles por el sensor, pero si las observaciones se realizan con una frecuencia alta el agente puede llegar a impactar negativamente el tiempo de respuesta del sistema.

- Considerar el *volumen de los datos, ¿*qué tantas observaciones deben ser registradas para identificar un síntoma?, este es un aspecto importante del diseño del agente que depende de la naturaleza de los atributos observados; hay síntomas que son claramente identificables en una variación abrupta del valor de un atributo, pero dadas ciertas circunstancias, una variación abrupta podría ser el comportamiento normal del atributo, es así que solo mayor cantidad de datos "históricos" reflejan un verdadero síntoma.
- Considerar la *comprobación de síntomas* se refiere a identificar la señal o indicio de que algo está sucediendo o que va a suceder en un futuro. Esto implica para el diseñador del software conocer los atributos observados y sus relaciones para poder hacer un emparejamiento a síntomas.
- Considerar *aproximaciones de monitoreo dinámicas,* hace referencia a la facilidad proveída por el monitor para facilitar la autonomía, si se tiene un monitor que adapta su frecuencia y volumen de datos esto ayudaría a que el sistema minimice sobrecostos por una alta frecuencia de monitoreo y maximice la utilidad de los recursos de almacenamiento temporal.
- Considerar la *dependencia del contexto*, es importante tener en cuenta que la sensibilidad controla el proceso de adaptación por lo mismo los monitores deben poder acceder homogéneamente tanto al recurso de estado interno como a los recursos del entorno de operación en orden de que sirva a su propósito.

5.4 PROCESO DE DETECCIÓN

En esta sección se explora el agente de detección, luego del proceso de monitoreo se consigue un banco actualizado de síntomas, indicios de que algo está sucediendo o va a suceder, la función del agente de detección es proveer mecanismos para analizar *situaciones* para determinar si algún cambio necesita ser hecho. Es importante aclarar lo anterior, no se dice que estás *situaciones* se consideren como anómalas, por el contrario, si se reflexiona con cuidado podría darse paso a oportunidades de mejora, lo cual representa una gran ventaja para el sistema.

Figura 22. Agente de detección



Fuente. Elaboración personal

Como se aprecia en la imagen de la Figura 22 el agente de detección se apoya de los síntomas identificados en el monitoreo, y genera un aviso de cambio, si ciertas condiciones se cumplen. Por dar un ejemplo, una solicitud de cambio puede ocurrir cuando el agente determine que alguna política no ha sido cumplida.

En muchos casos, el diseñador se percata que el comportamiento del sistema observado es complejo y puede emplear técnicas de predicción avanzadas tal como: series de tiempo, modelos de consulta, o reconocimiento de patrones.

Algunas consideraciones a la hora de llevar a cabo el diseño de este agente se presentan a continuación:

- Considerar la *influencia del módulo de conocimiento*, el análisis para la toma de decisiones está fuertemente influenciado por el conocimiento almacenado en el agente, pues este contiene los parámetros de comparación de este.
- Considerar el *establecimiento de políticas o reglas*, una política es una medida que se adopta para dirigir las situaciones que afectan al sistema en estudio o que guardan relación con ella. Así a partir del estudio de diversos síntomas en el aplicativo, el agente puede disparar un evento de cambio.
- Considerar la *correlación de síntomas,* una buena detección debe considerar la correlación entre variables, un ejemplo de ello puede ser el siguiente. Considere que un módulo compuesto por una cola de mensajes y tres hilos de atención, en el escenario en que se ha tenido un volumen de mensajes en cola promedio de 5 mensajes. Imagine que un monitor reporta el síntoma "*Abruptamente el volumen de mensajes se incrementa a 15 unidades*", podría pensarse que la solución más lógica es elevar el evento: "*incrementar la capacidad atención*" lo cual implicaría un incremento en el número de hilos. Ahora, si hay otro monitor para los hilos que reporta el síntoma de: "*Dos hilos han sido bloqueados*". Con la consideración de este nuevo síntoma, lo más sensato, sería elevar un evento de "*Recuperación de elementos de atención*". Incluso si se añaden nuevos síntomas se puede en pensar en llegar a identificar la causa real del incremento en el volumen de la cola de mensajes.
- Considerar los *eventos al detectar cambios*, la idea de informar cambios mediante eventos tiene muchas ventajas, entre ellos la separación entre

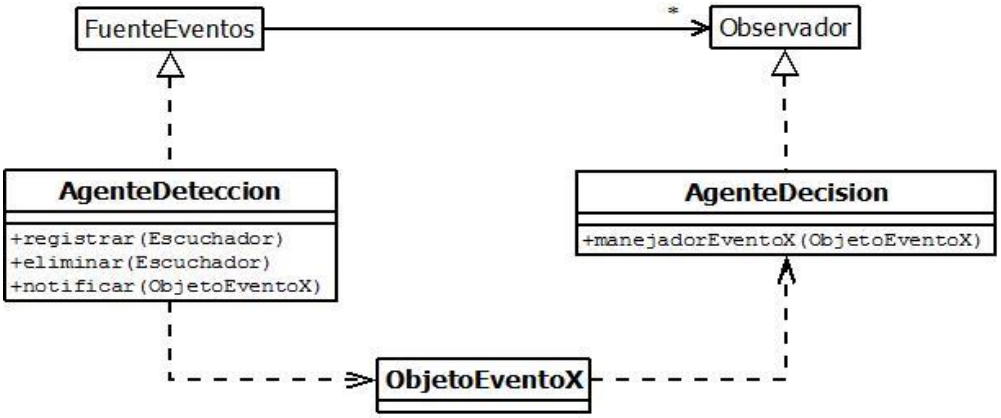
la detección y las acciones para manejarlo, sin embargo, es el diseñador del programa quien debe definir claramente qué eventos tendrá su programa y la información que cada uno debe compartir para que los agentes de decisión puedan manejarlos apropiadamente. Otro tema interesante es que los agentes de decisión pueden escuchar o no los eventos que deseen. Así un evento generado por la ramificación θ_i de la estructura jerárquica puede ser manejado por muchos agentes de decisión con propósito diferentes.

- Considerar el *nivel de los parámetros*, un evento de solicitud de cambio describe las modificaciones que el componente de detección considere necesarias o deseables, dicha consideración se examina frente a la luz de parámetros del sistema, en las consideraciones de diseño de este agente es bueno tener en cuenta que dichos parámetros entran en tres categorías, parámetros globales, locales y operativos. Siendo de esta manera puede que un síntoma impacte un parámetro operativo, pero no afecte, de forma significativa el comportamiento del sistema y por lo tanto no se vea reflejado en los parámetros locales y globales del mismo.

5.4.1 Interacción con agentes de decisión

Como fue mencionado como salida de este agente se tienen eventos que serán manejados por agentes decisión, la propuesta de este marco para la interacción entre estos agentes es emplear un patrón de diseño como el observador, donde el *agente de detección* juega el rol de *fuente de datos*, mientras el agente de decisión juega el rol de observador, la imagen en la Figura 23 da mayor claridad a la estructura de dicho patrón aplicado al aspecto mencionado del marco de trabajo propuesto, se describe con mayor detalle cada elemento del diagrama a continuación.

Figura 23. Interacción entre agentes Detección/Decisión



Fuente. Elaboración personal

A continuación, se describen los participantes de la anterior relación de forma desglosada:

Fuente Eventos: proporciona una interfaz para registrar y eliminar observadores, además debe conocer todos los observadores registrados.

Observador: define el método que usa la fuente de eventos para notificar eventos de cambio.

Agente de detección (*Fuente de eventos concreta*): se encarga de realizar las operaciones ya mencionadas y notificar los cambios que son de interés para los observadores registrados.

Agente de decisión (*Observador concreto*): mantiene una referencia al agente de detección concreto e implementa la interfaz de actualización, es decir, guardan la referencia del agente que observan, así en caso de ser notificados de algún cambio, pueden preguntar sobre este cambio al agente de detección.

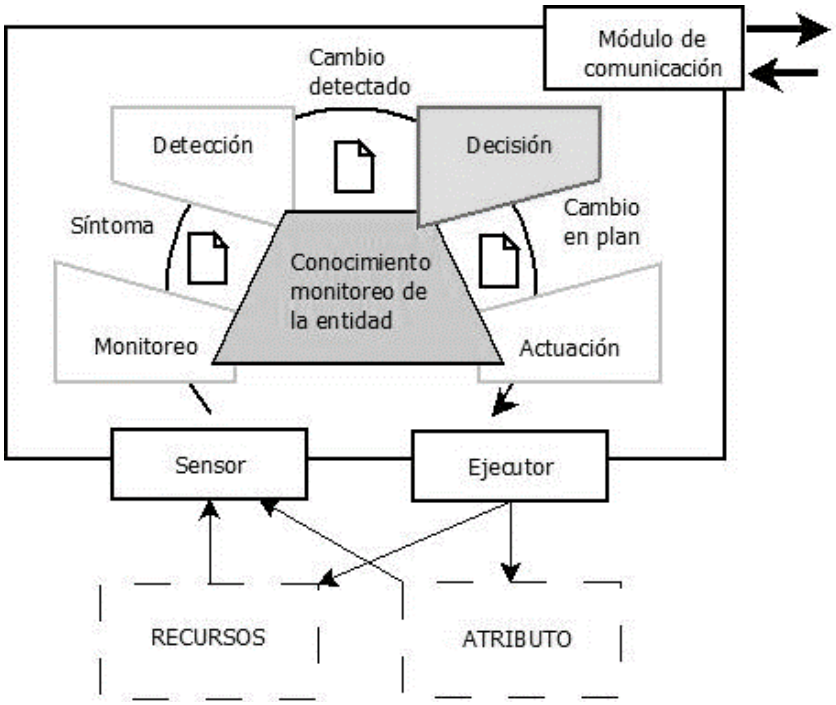
Objeto Evento *x*: cada evento puede acompañarse de un objeto que porte información sobre el evento, por decir algo, instante en el que ocurre, referencia al agente que lo genera, etc.

5.5 PROCESO DE DECISIÓN

Los agentes de decisión seleccionan un procedimiento para propagar una alteración deseada en el recurso(s) administrado(s). Las acciones de este agente pueden tomar muchas formas, desde la simple ejecución de un comando hasta un flujo de trabajo completo. Este agente se activa cuando captura un evento al que se ha registrado, entonces a partir de esta señal y de la información del mensaje adjunto en el evento genera un plan de cambio apropiado, dicho plan representa un conjunto de cambios deseados para el recurso que "*administra*", y lógicamente pasa esos cambios a la etapa de actuación.

La imagen de la Figura 24 presenta las entradas y salidas de la etapa que se enfatiza en dicho agente. Los agentes de decisión pueden llegar a realizar un análisis selectivo sobre la viabilidad de diferentes esquemas de cambio y la competencia entre ellos.

Figura 24. Agente de decisión



Fuente. Elaboración personal

Algunas consideraciones a la hora de llevar a cabo el diseño de este agente se presentan a continuación:

- Considerar las *restricciones dinámicas*, raras veces los límites de operación de sistema se comportan de forma constante, si ellos imponen restricciones que deben considerarse por los agentes de decisión es importante tener una caracterización de ellos e identificar cuáles de ellos cambian dinámicamente de esta manera los agentes de decisión pueden considerar una mayor diversidad de los esquemas de cambio.
- Considerar el *nivel de las metas*, una meta representa una finalidad a la que se dirigen las acciones o deseos del sistema o de sus elementos, ellas pueden ser divididas en grupos o niveles, y cada nivel beneficiar un subconjunto de atributos de calidad diferentes. El diseñador debe procurar crear niveles compatibles de metas, los niveles ayudan a incluir

una “*prioridad*” que puede marcar una tendencia del sistema a favorecer unos esquemas de cambio más de otros.

- Considerar las *acciones del agente*, se pueden seguir muchas acciones luego de una detección de un cambio, desde sugerir no hacer nada, al ajuste de algunos parámetros de operación, hasta sugerir una adaptación tan radical como interrumpir o eliminar un componente. Pero la principal función de este agente es sugerir “el mejor” esquema de cambios que considere pertinente.
- Considerar los *esquemas de cambio en competencia*, ¿qué conjunto de cambios pueden llevar a que el sistema tienda a un estado de mejora, o a beneficiar ciertos atributos de calidad, o a corregir ciertos comportamientos? La respuesta a esta pregunta es, no hay una única respuesta. Toda modificación puede desembocar en una catástrofe, sin embargo, la adaptación es un proceso paulatino y reversible, la única forma de conocer si el cambio surtió buen o mal efecto sin conocimiento previo es mediante la prueba-error-recompensa. La recompensa promueve la supervivencia o aniquilación de un esquema de cambio, así, que esta es una característica innata de la adaptación y debe ser tenida en cuenta por el agente de decisión. En resumen, hay diversos esquemas en competencia, y los resultados de la acción en el mundo real influyen de modo retroactivo en dicha competencia.
- Considerar la *publicación del plan de cambio*, ¿qué sucede una vez se escoge cambio a implantar? Si bien este tipo de agentes se registraron como escuchadores de eventos, qué pasa con su salida, su salida es un plan con instrucciones detalladas del cambio a realizar, pero ¿son tan detalladas como para considerarse instrucciones de codificación en el lenguaje de programación en específico? La respuesta es sí, básicamente en este caso se decidió realizar la instanciación de agentes que implementan microajustes, esto significa que, para cada tipo de cambio, por ejemplo, disminuir la cantidad de hilos de trabajo, se requerirá de un agente que monitoree el estado del sistema, detecte y

decida el instante en el cual puede realizar el cambio y finalmente ejecute las acciones para cumplir con la tarea asignada.

- Considerar la *discrepancia entre predicción y observación*: lo normal es que las predicciones comienzan siendo erróneas, pero si la secuencia tiene una estructura fácil de captar, la discrepancia entre predicción y observación hará que los esquemas erróneos sean descartados en favor de otros mejores y pronto prever con precisión cuál será la próxima imagen del sistema.

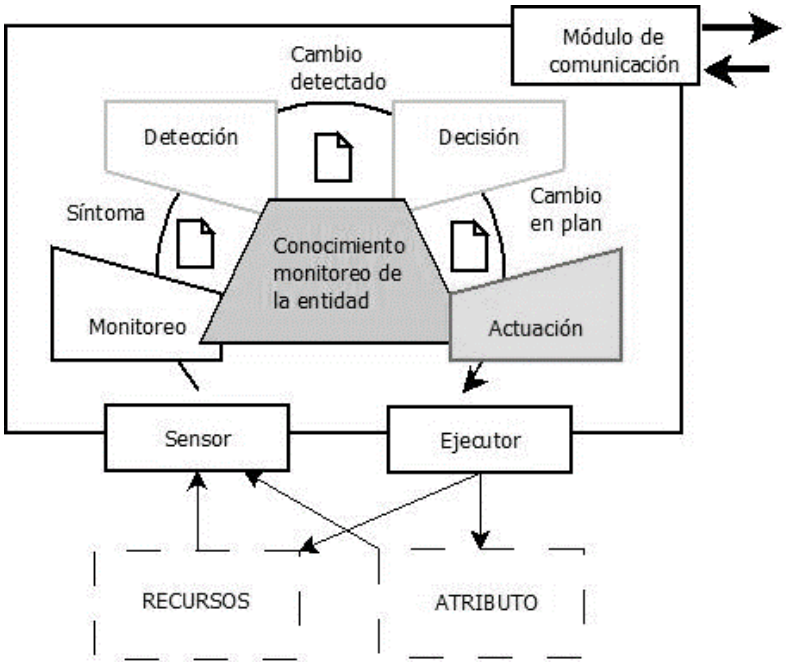
5.5.1 Interacción con agentes de actuación

Básicamente la única interacción requerida con los agentes de actuación es su creación y el envío de la señal de inicio de ejecución, lo cual implica la creación de una instancia soportada por un hilo de ejecución particular y la solicitud a dicho hilo de iniciar su trabajo.

5.6 PROCESO DE ACTUACIÓN

Los agentes de tipo actuación proveen mecanismos para la programación y realización de los “cambios” del *esquema de cambio* seleccionado por los agentes de decisión. Una vez generado un *plan de cambio* correspondiente a un evento de cambio; algunas acciones pueden necesitar modificar el estado de uno o más recursos administrados, es bueno resaltar que el cambio puede ser realizado no solo por un agente de actuación, sino por varios de estos.

Figura 25. Agente de actuación



Fuente. Elaboración personal

La imagen de la Figura 26 presenta las entradas y salidas de la etapa que se enfatiza en dicho agente. Los agentes de actuación pueden llegar a atender uno o varios de los cambios sugeridos en el esquema, todo depende, de los efectores que tenga a cargo.

Considere el caso hipotético en que se ha generado un plan de cambio para un aplicativo, consistente en una arquitectura C/S sin estado en el *backend*, y se presenta el siguiente escenario: una "*detección de un incremento en el volumen de clientes conectados"*, esto consecuentemente deriva en un evento tipo *"hacer distribución de carga de trabajo*", es posible que el subárbol relacionado a cada instancia *Cliente* recibe este evento pero no decida hacer nada, sin embargo, un agente de decisión del lado servidor, analiza entre sus esquemas de cambio y elabora el siguiente plan: 1) crear nueva instancia de servidor 2) crear instancia de distribuidor de carga 3) reconectar los componentes 4) reiniciar operación. Cuando este plan se comunica a los agentes de actuación varios de ellos pueden participar de la ejecución de este

plan. Desde agentes que garanticen la continuidad de la operación, hasta agentes encargados de crear y gestionar nuevos recursos, como el distribuidor de carga. Cabe mencionar que parte de la ejecución del cambio podría involucrar la actualización del conocimiento usado por el agente, sobre las características o comportamientos del recurso administrado.

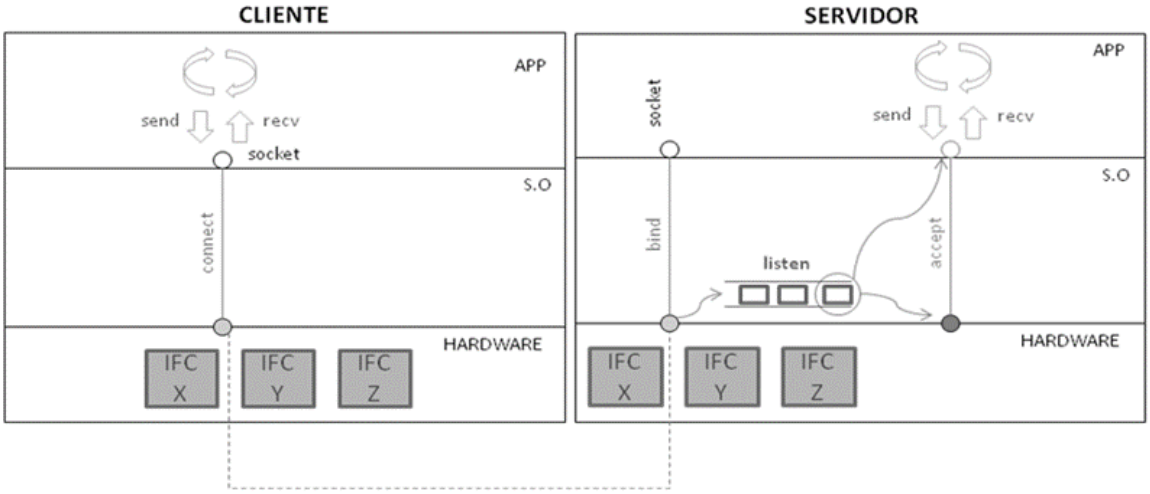
Algunas consideraciones a la hora de llevar a cabo el diseño de este agente se presentan a continuación:

- Considerar la *estrategia y la táctica,* la estrategia es un plan de acción que corresponde a la totalidad del esquema de cambio, responde a lo qué se debe hacer para mantenerse alineado con las metas de sistema y alcanzar la adaptación propuesta. La táctica, por su parte, plantea cómo se llevan a cabo los esquemas de cambio, el método que utilice el agente para alcanzar los objetivos se le conoce como táctica, el objetivo de la táctica es calcular con precisión cada acción, encontrar los agentes adecuados para responder al plan de cambio.
- Considerar la *propagación del cambio,* para redes de agentes de actuación que requieran comunicación entre ellos según el tipo de agente se propone que todos ellos puedan ver las recomendaciones en el plan cambios, es decir, *la estrategia*, y basado en algún conocimiento a priori sobre el elemento que operaran elijan *la táctica* para llevar a cabo los cambios sugeridos, si la granularidad de los recursos es fina, y se cuenta con agentes distintos para un grupo heterogéneo de elementos, posiblemente algunos agentes de actuación decidan aplicar efectores sobre su elemento administrado, mientras que otro no.
- Considerar la *continuidad de la operación, ¿*cuál es o será el estado de finalización de las operaciones actuales del elemento, sin afectación del resultado ni de la funcionalidad del sistema?

5.7 COMPONENTE EFECTOR

Los efectores son finalmente los métodos o procedimientos encargados de llevar a cabo los cambios, ellos dependen de las características de cambio del recurso específico administrado, por poner un ejemplo simple, considere el caso de un socket, es decir un punto de comunicación entre dos procesos en ejecución, la imagen de la Figura 27 resume los pasos para establecer dicha conexión.

Figura 26. Elementos de una conexión por sockets



Fuente. Elaboración personal

Recordando brevemente la API para manejar este recurso:

PRIMITIVA	SIGNIFICADO
socket	Crea un nuevo punto terminal de comunicación
bind	Adjunta una dirección local a un socket
listen	Anuncia la disposición a aceptar conexiones; indica el tamaño de la cola de conexiones
accept	Bloque al invocador hasta la llegada de un intento de conexión
connect	Intenta establecer activamente una conexión

send	Envío de datos a través de la conexión
receive	Recibe datos de la conexión
close	Libera la conexión.

En la imagen se presentan los elementos (recursos) empleados en las diferentes capas del sistema para soportar el uso de puntos de conexión. Si, por ejemplo, hubiera elementos efectores trabajando en la capa del hardware, este debería ofrecer mecanismos para aprovechar los recursos disponibles allí como una API o algo similar, de esta manera a través de este mecanismo se puede decidir utilizar una *interfaz física de conexión* particular o deshabilitar por no ser requerida. Con esto se hace referencia a la dependencia de las características de cambio del recurso.

- Considerar la *granularidad del cambio*, los cambios pueden ser de grano grueso, tal como reemplazar componentes completos o reorganizar las conexiones entre componentes, pero ellos pueden ser también de grano fino, tal como cambiar los parámetros operacionales, el estado interno o lógica de funcionamiento de componentes individuales. Naturalmente, el modelo de cambio para decisiones de adaptación tiene que ser creado después que el sistema exista.
- Considerar los *micro-ajustes*, aunque el término es muy de la mecánica de precisión y hace referencia a la forma en que dos piezas de una misma máquina se acoplan entre sí, se basa en la relación entre dos elementos, y las características de dicha relación. En el contexto de la adaptación es el conjunto de medidas encaminadas a reducir los" desequilibrios *existentes*" entre dos elementos en el sistema.

5.8 TEMAS TRANSVERSALES

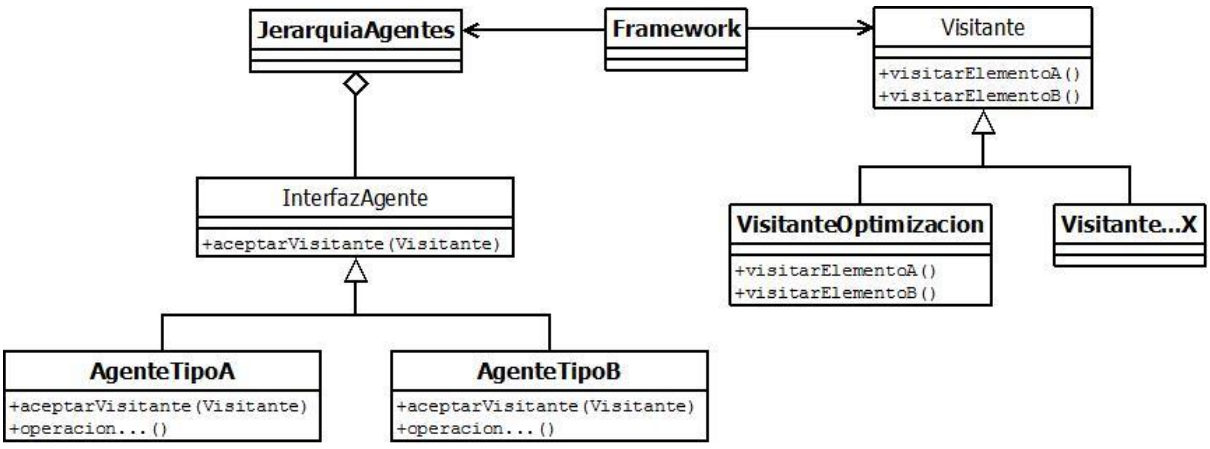
Los temas abordados en esta sección se han reunido como consideraciones generales de diseño del marco de trabajo, algunos aclaran aspectos estructurales o de comportamiento, mientras que otros abren discusiones para investigaciones futuras.

5.8.1 Visitantes especializados

Como se ha mencionado anteriormente el marco de trabajo sugiere trabajar con el patrón de diseño visitante, este es un patrón de comportamiento, que permite incluir nuevos métodos o procedimientos en un elemento de implementación sin tener que modificarlo, es por lo regular muy utilizado en compiladores, intérpretes y analizadores de código.

Debido a su flexibilidad es recomendable usar el patrón cuando se trabaja con estructuras jerárquicas (árboles), o cuando hay una amplia diversidad de elementos como es el caso del marco de trabajo propuesto dado que da posibilidad de ampliación e incluir nuevas funcionalidades.

Figura 27.. Patrón del visitante en marco de trabajo



Fuente. Elaboración personal

La imagen de la Figura 28 muestra los principales elementos del patrón aplicados al marco de trabajo y a continuación se realiza una descripción de estos:

Jerarquía de agentes: puede enumerar los agentes (organizarlos estructuralmente como sea conveniente) y puede proporcionar una interfaz de alto nivel para permitir al *Visitante* visitar sus elementos.

Visitante: una interfaz que declara una operación de visita para cada operación a efectuar a cada elemento de la estructura de agentes.

Visitante…X (*Visitante concreto*): Implementa cada operación declarada por la interfaz *Visitante*.

Agente (*Nodo*): define una operación que le permite aceptar la visita de un Visitante en específico, de esta manera algunos agentes aceptan unos visitantes, pero no otros.

AgenteTipo…X (*Agente concreto*): implementa la operación *aceptarVisitante* que se limita a invocar su correspondiente método del *Visitante*.

Con el anterior patrón introducido dentro del marco de trabajo propuesto, se heredan todas las ventajas del patrón:

- Es fácil añadir nuevas operaciones a la “Jerarquía de agentes”; sólo hay que crear un nuevo “Visitante”, en vez de cambiar todos los diferentes agentes a los que se quiere afectar.
- Hay una fuerte cohesión de funcionalidades dado que se juntan las operaciones relacionadas entre sí, separándose las que nada tienen que ver. Esto simplifica tanto los *Agentes* como los algoritmos definidos en los *Visitantes*.
- Se pueden recorrer jerarquías heterogéneas de agentes de distintos tipos, mientras que un *Iterador* tradicional no puede hacer esto.
- Uno de los aspectos más interesantes es que se puede ir acumulando el estado a medida que se recorre la estructura, en vez de tener que pasarlo como parámetro o guardarlo en variables globales.

5.8.2 Comportamiento emergente

Este concepto hace referencia a aquellas propiedades o procesos de un sistema no reducibles a las propiedades o procesos de sus partes constituyentes. El concepto de emergencia se relaciona estrechamente con los conceptos de auto-organización y superveniencia, y se define en oposición a los conceptos de reduccionismo y dualismo, y considera que "el todo, es más que la suma de las partes".

5.8.3 Comportamiento supervisado

La supervisión es la observación regular y el registro de las actividades que se llevan a cabo en el programa, es un proceso de recogida rutinaria de información sobre los aspectos de este. Supervisar es controlar qué tal progresan las actividades en consecuencia con las metas del sistema. Es observación sistemática e intencionada. De esta manera el diseñador puede crear un *Visitante* para realizar una especie de intervención o corrección del comportamiento, en otras palabras, se puede cambiar la emergencia por la imposición.

5.9 MÉTRICAS Y EVALUACIÓN

Calidad de servicio (QoS, por sus siglas en inglés) es posiblemente el nivel más alto para comparar sistemas modernos, este refleja en qué grado el sistema está alcanzando su meta primaria, QoS se compone típicamente de un número de métricas. La mayoría de las investigaciones de arquitecturas adaptativas buscan usar la autonomía de los agentes para mejorar el rendimiento (usualmente velocidad o eficiencia). Todas las métricas de QoS están ligeramente acopladas al área de aplicación o servicio del sistema y deben revisarse a la luz de este hecho, a continuación, se mencionan algunas consideraciones al respecto a las métricas de estos sistemas.

5.9.1 Costo de adaptación

En la actualidad el grado de este costo y su medición aún no está clara dada la heterogeneidad de diseños de estos sistemas. Actualmente la mayoría de los estudios de rendimientos de sistemas adaptativos basados en agentes luchan por su habilidad para alcanzar las metas del sistema. Sin embargo, los sistemas basados en agentes deben considerar el valor de las comunicaciones, las acciones realizadas durante el ciclo de adaptación, y costos de las acciones requeridas para alcanzar la meta. La comparación de sobrecostos es aún más complicada debido al hecho que la adición de autonomía significa adición de inteligencia, monitores y mecanismos de adaptación y estos cuestan.

5.9.2 Granularidad fina

Los componentes de grano fino con agentes de adaptación específicos adheridos a ellos serán altamente flexibles y quizá se adapten mejor a muchas situaciones, sin embargo, puede causar mayores sobrecostos en términos del sistema. Si se asume que cada agente se conecta a componentes de grano fino, este requiere datos del entorno y provee alguna forma de realimentación sobre su rendimiento, entonces potencialmente hay mayor flujo de información del entorno sobre el sistema global hacia el núcleo de conocimiento en cada agente. Evidentemente mayor sobrecosto que si la arquitectura de adaptación fuese centralizada.

5.9.3 Sensibilidad

Esta métrica se refiere a qué tan bien el sistema de adaptación se ajusta al entorno de operación en el cual está desplegado. Un sistema altamente sensible nota cambios sutiles cuando estos suceden y adapta (quizá sutilmente) para mejorarse a sí mismo basado en ese cambio. Sin embargo, en realidad siempre hay alguna forma de retraso en el *Feedback* de que algunas partes del entorno que han cambiado y esto impacta el tiempo de

respuesta del sistema adaptativo. Así, puede ser perjudicial para el sistema ser altamente sensible a su entorno, dado que potencialmente puede una avalancha constante de cambios de configuración.

5.9.4 Tiempo para adaptar y de reacción

Relacionado al costo de adaptación y a la sensibilidad, hay métricas concernientes a la reconfiguración del sistema. El tiempo para adaptación es la medida del tiempo que el sistema toma para ejecutar un cambio en el entorno. Esto es, el tiempo tomado entre la identificación de ese cambio, hasta que el cambio ha sido efectuado de forma segura y el sistema se vuelve a un estado confiable y de continuidad. El tiempo de reacción es el tiempo entre cuando un elemento del entorno ha cambiado y el sistema reconoce ese cambio, decide qué configuraciones son necesarias para reaccionar al cambio de entorno y tener el sistema listo para adaptarse.

5.9.5 Homeostasis

Otra métrica relacionada a la sensibilidad es la estabilización. Esto es, el tiempo tomado por el sistema para aprender de su entorno y estabilizar su operación. Es particularmente interesante para sistemas adaptativos abiertos que aprenden cómo mejorar la reconfiguración del sistema.

5.10 CONCLUSIONES

- El marco de trabajo propuesto tiene aún aspectos abstractos del conocimiento contenido y formato, y los mecanismos de algunos componentes del sistema, asegura que el conocimiento base contiene una representación del comportamiento del sistema y el entorno como se percibe por el sistema.
- El diseño del sistema técnico usualmente se enfoca en la intención de la funcionalidad del sistema y con frecuencia obedece al principio “*la*

función sigue al diseño". Consecuentemente el sistema es organizado en componentes que implementan las funcionalidades específicas de la aplicación. Sin importar el estilo arquitectónico elegido el marco de trabajo desarrolla ideas que pueden implementadas en cualquier arquitectura.

- Cuando se implemente un sistema específico de computación auto-adaptativa, se requiere proveer un modelo computacional que envuelva cada uno de los componentes, conectores e interacción entre los componentes, y entre el sistema y su entorno. Las ideas planteadas aquí tratan de cubrir este requisito.
- Un sistema basado en agentes como el propuesto ofrece una solución muy atractiva. Cada agente debe (1) encapsular un método de razonamiento específico a su tipo que implemente uno de los componentes del sistema y (2) proveer los mecanismos de interacción y comunicación con otros agentes.

6. CASO DE ESTUDIO

6.1 ELEMENTOS BASICOS

El objetivo de este capítulo es mostrar cómo aplicar el marco de trabajo propuesto a un programa, para lo cual el documento plantea el siguiente desarrollo, presentar el programa que será objeto de la adaptación, esto incluye funcionalidad y diseño, a continuación, se aplican todos los conceptos desarrollados en el último capítulo, mostrando la aproximación al diseño de un sistema complejo adaptativo.

Para el sistema considerado, se tiene que el servidor presta un único servicio, que consiste en una multiplicación matricial, que es una operación simple en el álgebra matricial. La razón por la cual se eligió esta operación es permitir variar el tiempo de respuesta del servidor en función de los tamaños de matriz recibidos en el mensaje de la solicitud. Un ejemplo de un mensaje de petición sería: "134, 254, 354", esta cadena reducida sería procesada por en el servicio y dicho mensaje se interpretaría como la solicitud de la operación $(\mathbf{134}, \mathbf{254})$ $\mathbf{x}$ $\boldsymbol{B}(\mathbf{254},\mathbf{354})$, de esta manera, el servicio construirá dinámicamente dos matrices e inicializará cada celda con valores aleatorios. Finalmente, un hilo trabajador realizara la operación del producto matricial y retornara al cliente la respuesta de la operación, es decir, para el ejemplo expuesto una matriz resultado con dimensiones $\boldsymbol{C(134, 354)}$ **.**

6.1.1 Producto matricial

No está de más, recordar en qué consiste un producto matricial. La primera condición que hay que verificar para poder multiplicar dos matrices, es que el número de columnas de la primera matriz debe coincidir con el número de qué filas la segunda. La matriz resultante del producto matricial quedará con el

número de filas de la primera matriz y con el mismo número de columnas de la segunda matriz. Es decir, si tenemos una matriz *2x3* y la multiplicamos por otra de orden *3x5*, la matriz resultante será de orden *2x5*. Hay muchas maneras de llevar a cabo esta operación, el método utilizado en este trabajo se describe a continuación.

Supongamos que *A = (aij)* y *B = (bij)* son matrices tales que el número de columnas de *A* coincide con el número de filas de *B*; es decir, *A* es una matriz de dimensión *mxp* y *B* una matriz de dimensión *pxn*. Entonces el producto *C=AxB* es la matriz *mxn* cuya entrada C*ij* se obtiene multiplicando la fila i de *A* por la columna j de *B*.

Esto es,

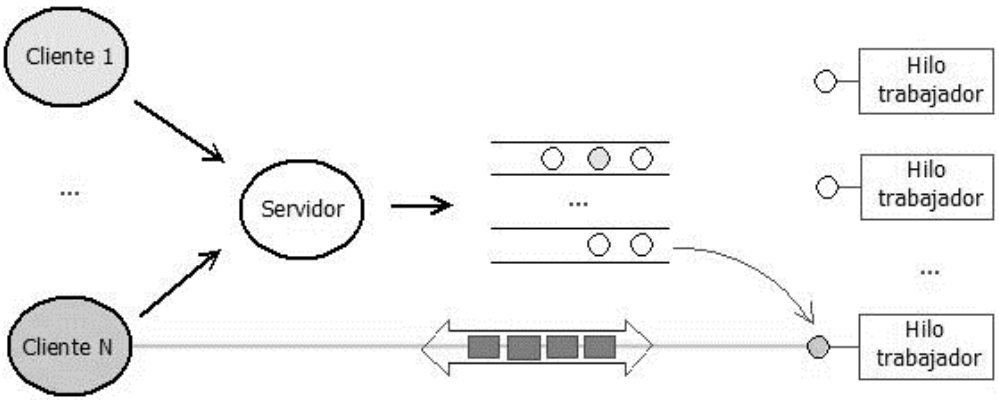
$$\begin{pmatrix} a_{11} & \dots & a_{1p} \\ \vdots & \dots & \vdots \\ a_{i1} & \dots & a_{ip} \\ \vdots & \dots & \vdots \\ a_{m1} & \dots & a_{mp} \end{pmatrix} \begin{pmatrix} b_{11} & \dots & b_{1j} & \dots & b_{1n} \\ \vdots & \dots & \vdots & \dots & \vdots \\ \vdots & \dots & \vdots & \dots & \vdots \\ \vdots & \dots & \vdots & \dots & \vdots \\ b_{p1} & \dots & b_{pj} & \dots & b_{pn} \end{pmatrix} = \begin{pmatrix} c_{11} & \dots & c_{1n} \\ \vdots & \dots & \vdots \\ \vdots & c_{ij} & \vdots \\ \vdots & \dots & \vdots \\ c_{m1} & \dots & c_{mn} \end{pmatrix}$$

donde $c_{ij} = a_{i1}b_{1j} + a_{i2}b_{2j} + \dots + a_{ip}b_{pj}$

6.1.2 Diseño del caso de estudio

La imagen de la Figura 29 presenta el programa que será sometido al marco de trabajo propuesto, esta sección hace una descripción básica del diseño de tal programa, se analizará las características para extraer información del diseño y se expondrá oportunidades de cambio y adaptación, el principal objetivo es comprender este programa para pasar a aplicar el marco de trabajo, así que se revisa sus elementos constituyentes y analiza alguna información de estos que puede ser útil en el ciclo de vida de adaptación.

Figura 28. Elementos de la aplicación a ser adaptada



Fuente. Elaboración personal

Este programa tiene una arquitectura *Cliente/Servidor*, la imagen se ha cargado más con aspectos del servidor que del cliente. El módulo *servidor*, cuenta con un hilo principal para la atención de solicitudes de conexión, un *pool* de colas de conexión y finalmente un pool de hilos de trabajo, estos últimos interpretan los mensajes de operación enviados por los clientes y ejecutan la operación principal ya descrita.

El *hilo de atención* recibe todas las solicitudes de conexión del cliente, pero le es imposible atender directamente cada conexión, debido a que esto bloquearía el programa, en su lugar, la tarea principal de este hilo es atender la solicitud y poner en una cola seleccionada bajo algún criterio dicha conexión, esta estructura de datos hace las veces de una cola de espera. Cada vez que un *hilo trabajador* queda libre de su carga de trabajo, este puede revisar qué cola tiene una conexión pendiente, extraer una conexión de la cola y pasar a ejecutar el trabajo requerido por el cliente.

Como el protocolo empleado es TCP/IP y este requiere una conexión dedicada, cada hilo puede atender únicamente un cliente a la vez, sin embargo, se presenta con frecuencia que un hilo entra en un estado de detención temporal por múltiples razones entre ellas, la espera de un recurso

del sistema, por lo cual es muy útil que otro hilo utilice estos tiempos “muertos” del proceso y lo emplee para adelantar otras tareas. En ocasiones tener varios hilos mejora el tiempo de respuesta de la aplicación, pero se debe ser cuidadoso al tener muchas instancias de hilos ya que estos pueden comenzar a interferir con otros e impactar negativamente el rendimiento de la aplicación.

Por su parte, el módulo *cliente* se encarga exclusivamente de establecer una conexión con el servidor y enviar peticiones de operación, cuyo mensaje contiene las dimensiones de las matrices a operar.

6.1.3 Aspectos dinámicos del programa

Para entrar un poco en materia comencemos analizando en detalle la estructura y comportamiento del programa propuesto, para ver las oportunidades de explotación en el marco de trabajo para adaptación.

La primera consideración valiosa es la identificación de los elementos en el aplicativo que pueden considerarse recursos, para ello se expone la siguiente definición de lo que puede considerarse un recurso:

> *Los recursos son contenedores de colecciones de atributos, estos atributos pueden estáticos o dinámicos, y pueden obtenerse de la lectura directa de propiedades o mediante cálculos elaborados*[78]*.*

Bajo esta definición es posible identificar del diagrama presentado los siguientes atributos: *número de conexiones, conexiones por periodo de tiempo, tiempo medio entre conexiones, histograma de conexiones, tiempo medio de espera por cola de conexiones, tiempo transcurrido desde la última conexión, cantidad de colas de conexión, capacidad de cada cola de conexiones, número de conexiones total, número de conexiones en cola, disponibilidad por cola,*

[78] MORIN, Brice, et al. Unifying runtime adaptation and design evolution. En *Computer and Information Technology, 2009. CIT'09. Ninth IEEE International Conference on*. IEEE, 2009. p. 104-109.

disponibilidad total, cantidad de hilos ocupados, tipo de estrategia (hilo x cola, hilo x conexión, hilo x solicitud), cantidad de hilos libres, cantidad total de hilos, cantidad de mensajes atendidos por conexión, cantidad de mensajes atendidos por hilo, cantidad total de mensajes atendidos, frecuencia media de llegada de mensajes, tiempo medio de atención de mensajes, tiempo menor de procesamiento, tiempo mayor de procesamiento, tiempo medio de mensaje en cola por hilo trabajador, cantidad de mensajes por conexión, cantidad media de mensajes por conexión.

La anterior lista cubre alrededor de 27 atributos que pueden consultarse o calcularse sólo considerando los elementos del programa servidor, vale la pena aclarar que algunos de ellos son linealmente dependientes, es decir, que pueden obtenerse como combinación lineal de otros, esto se tiene en cuenta en etapas posteriores del proceso.

El siguiente paso es agruparlos en colecciones que llamadas recursos, es bueno aclarar que solo se están considerando recursos internos del sistema, aún queda por describir los recursos externos que también deben ser medidos.

Luego del paso anterior, se puede abstraer información de dichos recursos. Básicamente el proceso de adaptación tiene por finalidad modificar o ajustar el comportamiento y estructura del programa en función de esta información. Y para cumplir con este objetivo hay un proceso de detección, los métodos empleados para este proceso pueden variar, desde simples comparaciones con constantes de referencia, hasta análisis de tendencia de datos, métodos estadísticos o análisis multivariado que permita conocer las correlaciones entre los datos.

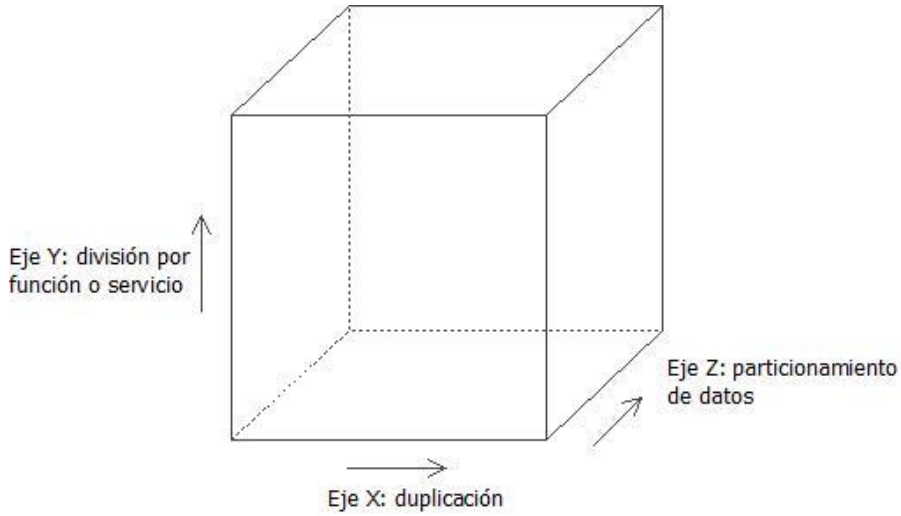
El paso anterior facilita la identificación de comportamientos en los recursos a esto viene un proceso de análisis de estos comportamientos para decidir si

una combinación de ello conduce o no a: situaciones anómalas, oportunidades de mejora, etc.

Una vez tomada la decisión de ejecutar un cambio, se sigue un procedimiento que puede ajustar los recursos, e incluso realizar modificaciones más significativas que se consideran pueden impactar positivamente el estado del programa.

Por ejemplo, si la carga de trabajo aumenta de manera significativa, se puede reaccionar realizando cambios que impliquen alterar la escalabilidad del sistema. La escalabilidad se entiende como la capacidad de respuesta de un sistema respecto al rendimiento y supone un factor crítico en el crecimiento de este.

Figura 29. Cubo de escalabilidad



Fuente. Elaboración personal

La imagen de la Figura 30[79] presenta los tipos de escalabilidad que puede tener un aplicativo de software: escalar en X consiste en ejecutar varias copias

[79] Abbott, Martin L., and Michael T. Fisher. *The art of scalability: Scalable web architecture, processes, and organizations for the modern enterprise*. Pearson Education, 2009.

de una aplicación detrás de un balanceador de carga; con N, en teoría copias cada una se encargaría de una fracción de 1/N de la carga. El escalar en Z, similar a X se tiene múltiples instancias idénticas de la aplicación, pero cada instancia es responsable por un subconjunto de los datos. Finalmente, a diferencia de los anteriores, que consiste en la ejecución de múltiples copias idénticas de la aplicación, el escalamiento en Y divide la aplicación en múltiples y diferentes servicios, cada servicio es responsable de una o más funciones estrechamente relacionadas.

Las siguientes secciones presentan la aplicación del marco de trabajo al aplicativo descrito.

6.2 ETAPA DE MONITOREO

6.2.1 Definición de recursos a monitorear

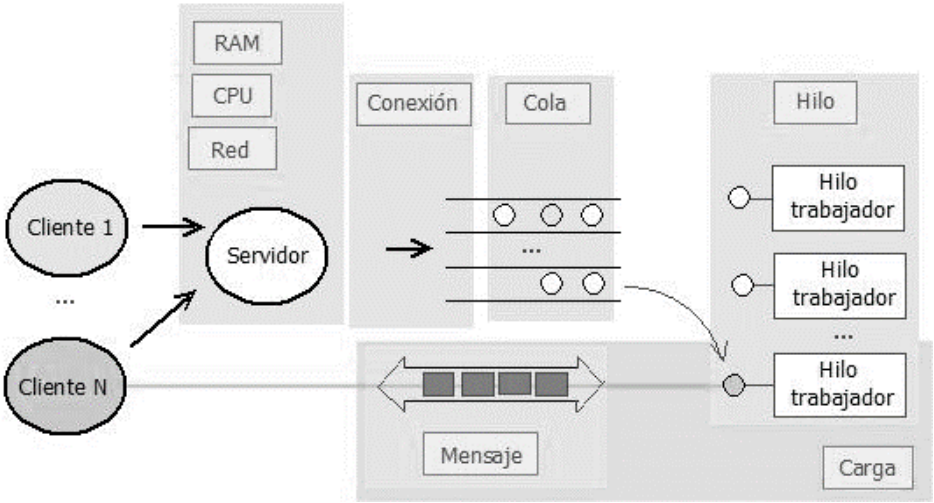
Al iniciar con el proceso de aplicación del marco de trabajo, se subdividen los atributos en conjuntos para formar recursos, recuerde que los recursos son colecciones, no necesariamente disjuntos de atributos.

Recursos internos	
Conexión	Número de conexiones, cantidad de conexiones por minuto, media de tiempo transcurrida entre conexiones (milisegundos), tiempo transcurrido desde la última conexión (milisegundos), número de conexiones en colas de espera.
Cola	Número de colas de espera activas, capacidad de cada cola de conexiones, disponibilidad de cada cola, tiempo medio de espera de atención por cola de conexiones.

Hilo	Cantidad de hilos ocupados, cantidad de hilos libres, media de mensajes atendidos por hilo, tiempo más bajo de atención, tiempo más bajo de atención, cantidad hilos bloqueados.
Carga	Tiempo medio de atención de mensajes, tiempo menor de procesamiento, tiempo mayor de procesamiento.
Mensaje	Cantidad total de mensajes atendidos, cantidad de mensajes por conexión, cantidad media de mensajes por conexión
Recursos externos	
CPU	Número de núcleos, disponibilidad por núcleo
Memoria	Usada, memoria de intercambio.
Red	Ancho de banda, promedio de ancho de banda

Como se aprecia en la tabla anterior, los recursos identificados son: *conexión, hilo, cola, carga y mensaje*; estos se han llamado *recursos internos* y como lo indica su nombre representan colecciones de atributos de elementos internos del sistema. El otro grupo de atributos en los que hay interés de medir son los correspondientes a los *recursos externos* y que son colecciones de atributos del entorno del sistema, estos completan el conjunto total de recursos: CPU, Memoria, y Red. La imagen de la Figura 31 aclara sobre cuales elementos del aplicativo trabajan los diferentes recursos identificados.

Figura 30. Recursos organizados



Fuente. Elaboración personal

Si bien la tabla presenta un listado extensivo de recursos, es necesario aclarar que para la prueba piloto se determinó no considerar algunos de ellos, por ejemplo, el recurso externo de red, por tratarse de una primera aproximación a la implementación. Por otra parte, el lector puede considerar muchos más atributos para los recursos internos, algunos de ellos se omitieron por ser linealmente dependientes, por ejemplo, la cantidad total de hilos es linealmente dependiente, al resultar de la suma entre los hilos ocupados y los hilos libres. Otros, se omitieron por simplificación, por ejemplo, los atributos relacionados al recurso mensaje, la principal razón radica en que los elementos allí citados también pueden ser combinación lineal de atributos en otros recursos.

Recuerde que para cada una de las etapas implica la construcción de uno o varios agentes especializados, cada paso de dicha construcción será descrito detalladamente.

6.2.2 Diseño del agente de monitoreo

El agente de monitoreo es aquel encargado de observar el estado de los recursos y resumir dichas observaciones en posibles síntomas. Esta etapa

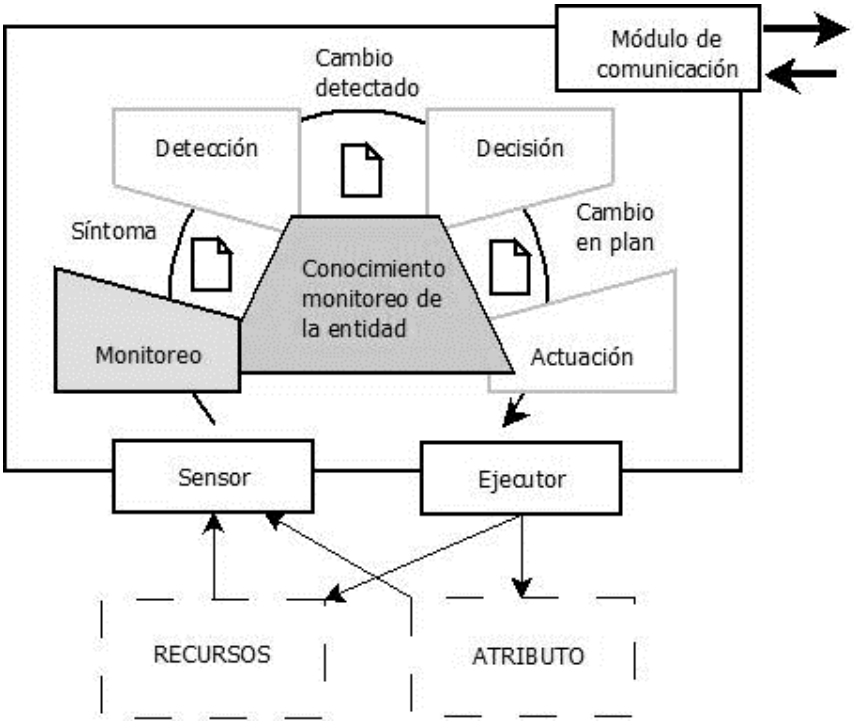
tiene como entrada de datos, las salidas de los agentes encargados de registrar mediciones de los diferentes *recursos: conexión, hilo, cola, carga, mensaje, y entorno*. A partir de los anteriores recursos, se proponen lo síntomas (vale aclarar que no son los únicos síntomas concebibles, el lector puede en su conocimiento y experiencia identificar otros):

- Incremento en el número de conexiones
- Incremento en el número de mensajes
- Bloqueo de hilos trabajadores
- Tiempos excesivos de atención de solicitudes
- Cargas de trabajo no equitativas entre hilos trabajadores
- Pico de procesamiento
- Utilización de excesiva de memoria RAM
- No disponibilidad de red
- Bloqueo frecuente de trabajos
- Promedio de tiempo de espera fuera de los límites permitidos
- Degradación del ambiente

Estos son algunos síntomas que pueden derivar en un estado anómalo de la aplicación. La primera tarea será entonces diseñar agentes que, ante ciertas combinaciones en las mediciones tomadas de los recursos del sistema, puedan extraer estos síntomas.

A continuación se recuerda la estructura de un agente de monitoreo:

Figura 31. Agente de monitoreo



Fuente. Elaboración personal

Para la prueba piloto, la atención se enfocará en los siguientes cinco síntomas:

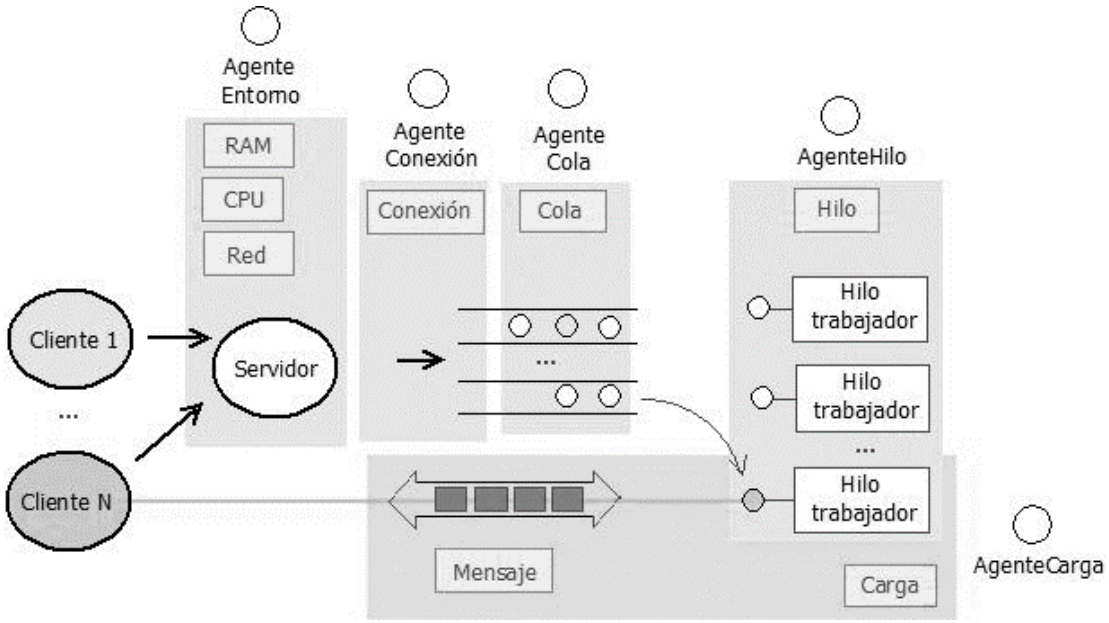
- Incremento en el número de conexiones
- Bloqueo de hilos trabajadores
- Tiempos excesivos de atención de solicitudes
- Cargas de trabajo no equitativas entre hilos trabajadores
- Degeneración del ambiente

Para iniciar su diseño se sugiere considerar la anterior lista de síntomas y la imagen en la Figura 32 a partir de la cual se sugiere el diseño del mismo. La estrategia de monitoreo propuesta es el *Monitoreo activo* el cual implica modificar el software en algún nivel para obtener cierta información del entorno o de sistema. Es conveniente recordar brevemente las consideraciones (restricciones) para el diseño en estas instancias: *parámetros de la aplicación*,

frecuencia de monitoreo, cantidad de las muestras tomadas, comprobación de síntomas, aproximaciones al monitoreo dinámico, dependencia del contexto.

Para el grupo de síntomas que se quieren identificar se propone tener los agentes mostrados en la imagen de la Figura 33.

Figura 32. Agentes de monitoreo



Fuente. Elaboración personal

La idea principal es que los agentes referidos puedan identificar los síntomas mencionados, si bien la gráfica presenta un agente por recurso, no es una imposición de diseño, otro diseñador puede sugerir un único agente para la captura de todos los síntomas. Es bueno recordar que todos estos agentes son de monitoreo, pero enfocan en identificar síntomas de diferentes partes del aplicativo. Cada agente mantendrá en su núcleo de conocimiento información valiosa respecto al(los) síntoma(s) identificados, facilitando al agente de detección realizar consultas para obtener más información al respecto.

6.2.3 Parámetros de la aplicación

Un primer paso para aproximarse al diseño de estos agentes es considerar los parámetros definidos para la aplicación, se ha decidido presentar tales parámetros en una de tres categorías, parámetros globales, parámetros locales y parámetros operativos, resumidos a continuación:

Parámetros globales	*Número de procesos servidor, número máximo de conexiones, tiempo máximo de atención por conexión, cantidad máxima de mensajes atendidos por conexión antes de poner la conexión en cola de espera nuevamente.*
Parámetros locales	*Número de hilos, número de colas, número máximo de colas, número mínimo de colas, número máximo de hilos, número mínimo de hilos.*
Parámetros operativos	*Cantidad máxima de hilos asignados a una única cola, tamaño máximo de colas de espera, tiempo máximo de procesamiento por hilo.*

Los valores para estas variables son asignados a voluntad por el diseñador e impactan el desempeño agente en cada etapa. Para la etapa en cuestión la manera en la cual se ve impactado el diseño del agente es que los procedimientos de toma de mediciones, adelantados por los agentes, pueden tardar más o menos tiempo dependiendo de la variación del valor de alguno de ellos.

Por dar un ejemplo, considere un establecimiento inicial en número de hilos, evidentemente los accesos a memoria pueden provocar tiempos de detención en los hilos trabajadores, si el valor inicial establecido para este parámetro es de 1, con seguridad se presentarán momentos en que el hilo se vea detenido

por accesos a recursos “más lentos”, un incremento en una unidad, puede mejorar significativamente el tiempo de promedio de respuesta a las solicitudes, debido a que mientras un hilo se encuentra bloqueado, otro puede utilizar este tiempo muerto del procesador para adelantar su tarea. Aunque para este caso particular la respuesta al incremento es una mejora en el tiempo de respuesta, no siempre será así. Considere una situación donde los hilos comparten un recurso en común, por ejemplo, un método de operación de suma de matrices, si de dicho método se conserva una única instancia, este se volverá un recurso compartido y si el número de hilos crece significativamente, ellos se bloquean hasta poder acceder a este recurso, impactando negativamente el rendimiento de la aplicación por la adición extra en la lógica de administración de hilos.

6.2.4 Síntomas contra restricciones

La siguiente tabla realiza un cruce entre síntomas que se van a identificar y restricciones mencionadas previamente, la idea es establecer un valor inicial para que permitan ir dando forma a dichos agentes:

Incremento en el número de conexiones	
Frecuencia de monitoreo	*Conexiones/Segundo*
Cantidad de muestras	*Últimos 50 registros de conexión.*
Comprobación de síntomas	*Verificar una tendencia al alza en el número conexiones.*
Monitoreo dinámico	*No*
Dependencia del contexto	*Recurso: Conexión*

Bloqueo de hilos trabajadores	
Frecuencia de monitoreo	*Cada hilo/Minuto*

Cantidad de muestras	*Estado actual por hilo*
Comprobación de síntomas	*Verificar el estado de los hilos.*
Monitoreo dinámico	*Si. En caso de presentarse síntoma, reprogramar la frecuencia de monitoreo a (Cada hilo) /20 Segundos. En otro caso a c/u Hilo/Minuto*
Dependencia del contexto	*Recurso: Hilo*

Tiempos excesivos de atención de solicitudes	
Frecuencia de monitoreo	*Cada cola/Minuto*
Cantidad de muestras	*Últimos 50 registros con información del valor promedio de atención por cola.*
Comprobación de síntomas	*Comprobar que hay una tendencia al alza en el valor promedio en tiempo de atención.*
Monitoreo dinámico	*No*
Dependencia del contexto	*Recursos: Entorno, Cola*

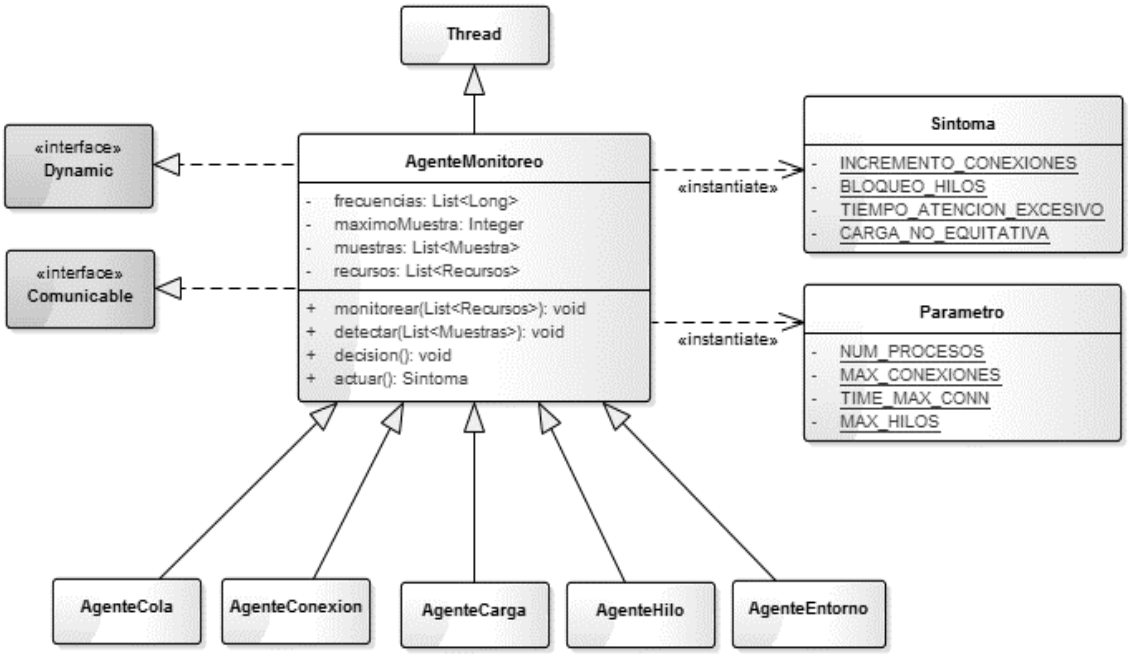
Cargas de trabajo no equitativas entre hilos trabajadores	
Frecuencia de monitoreo	*Cada hilo/Minuto*
Cantidad de muestras	*Las últimas 20 atenciones por hilo.*
Comprobación de síntomas	*Verificar media, y desviación estándar para cada hilo sobre los datos y posteriormente comparar revisar la dispersión de los datos.*
Monitoreo dinámico	*No*
Dependencia del contexto	*Recursos: Carga*

Degeneración del ambiente	
Frecuencia de monitoreo	*Cada hilo/Minuto*

Cantidad de muestras	*Ultimas 50 lecturas de memoria de intercambio y uso de CPU*
Comprobación de síntomas	*Verificar un incremento en la utilización de los núcleos del sistema, un incremento en el uso de la memoria de intercambio.*
Monitoreo dinámico	*Si. En caso de presentarse síntoma, reprogramar la frecuencia de monitoreo del entorno cada 10 Segundos. En otro caso mantener la frecuencia de muestreo por minuto.*
Dependencia del contexto	*Recursos: Entorno*

La siguiente gráfica muestra un diseño técnico del agente de monitoreo:

Figura 33. Diseño de agentes de monitoreo



Fuente. Elaboración personal

Básicamente se tiene un clasificador genérico que contiene los atributos que manejan las restricciones ya mencionadas que implementa la interfaz *Dynamic* para variar aspectos propios de las restricciones, y la interfaz *Comunicable* para permitir consultar información detallada de los síntomas, el objetivo de estos agentes es convertir las muestras tomadas a los recursos en objetos de tipo síntoma que puedan ser posteriormente consultados por otros agentes y hacer tratamiento de los mismos, las instancias de la clase síntoma son solo objetos de marcación, únicamente empleados para su identificación. A continuación, se explica brevemente cada uno de los métodos y su responsabilidad a este nivel, contrastados contra el tipo de agente concreto implementado.

AgenteConexion

Monitorear: Se toma muestras del recurso de conexión: número de conexiones, cantidad de conexiones por minuto, media de tiempo transcurrida entre conexiones (milisegundos), tiempo transcurrido desde la última conexión (milisegundos), número de conexiones en colas de espera.

Detectar: Se encarga de analizar los últimos 50 registros y comprobar la tendencia de los datos, se puede utilizar desde simples comparaciones incluso algoritmos más elaborados como una regresión lineal, para la prueba se opta por esta última.

Decidir: Se activa una alarma interna de síntoma, en caso en que se detecta una tendencia al alza en el número de conexiones se realiza la actuación, un decremento en la media de tiempo transcurrida entre conexiones y un alza en el número de conexiones en colas de espera.

Actuar: En este caso se genera un síntoma con identificador "*Incremento en el número de conexiones*" y se almacena dicha ocurrencia como registro histórico del evento.

AgenteCola

Monitorear: Se toma muestras del recurso de cola: número de colas de espera activas, capacidad de cada cola de conexiones, disponibilidad de cada cola, tiempo medio de espera de atención por cola de conexiones.

Detectar: Se encarga de analizar los últimos 50 registros y comprobar la tendencia de los datos, se puede utilizar desde simples comparaciones incluso algoritmos más elaborados como una regresión lineal, para la prueba se opta por esta última para cada dato registrado de las muestras.

Decidir: Se activa una alarma interna de síntoma, en caso en que se detecta una tendencia a la baja en la disponibilidad media de las colas, un alza en el tiempo medio de espera de atención por cola número y comparar si los datos descriptivos en términos de cantidad y capacidad actuales superan el promedio histórico.

Actuar: En este caso se genera un síntoma con identificador "*Tiempos excesivos de atención de requerimientos*" y mantenerlo como registro histórico.

AgenteHilo

Monitorear: Se toma muestras del recurso de hilo: Cantidad de hilos ocupados, cantidad de hilos libres, media de mensajes atendidos por hilo, tiempo más bajo de atención, tiempo más bajo de atención.

Detectar: Se encarga de analizar los últimos 50 registros y comprobar la tendencia de los datos, se puede utilizar desde simples comparaciones incluso algoritmos más elaborados como una regresión lineal, para la prueba se opta por una combinación de las dos.

Decidir: Se activa una alarma interna de síntoma, en caso en que se detecta una tendencia al alza en el número de hilos bloqueados, y la relación de hilos ocupados sea mayor que la de hilos libres detectando también una reducción en la efectividad de atención, es decir, una tendencia al alza en el tiempo más bajo de atención.

Actuar: En este caso se genera un síntoma con identificador "*Bloqueo de hilos trabajadores*" y mantenerlo como registro histórico.

AgenteCarga

Monitorear: Se toma muestras del recurso de carga: tiempo medio de atención de mensajes, tiempo menor de procesamiento, tiempo mayor de procesamiento.

Detectar: Se toma las últimas 20 atenciones de los diversos hilos, se puede aplicar algo de estadística descriptiva sobre las muestras de datos. Aparte de esto se puede comparar con tendencias de crecimiento del tiempo medio.

Decidir: Se activa una alarma interna de síntoma, en caso en que se detecta una tendencia al alza en el tiempo medio de atención de mensajes y se presenta una dispersión alta en los datos de la muestra, además si la diferencia entre el menor y mayor tiempo de procesamiento es tan alta como para considerarla como un síntoma.

Actuar: En este caso se genera un síntoma con identificador "*Cargas de trabajo no equitativas entre hilos trabajadores*" y mantenerlo como registro histórico.

AgenteEntorno

Monitorear: Se toma muestras del recurso de carga: número de núcleos del sistema, carga de trabajo por núcleo, memoria utilizada, memoria de intercambio.

Detectar: Se encarga de analizar los últimos 50 registros y comprobar la tendencia de los datos, se puede utilizar desde simples comparaciones incluso algoritmos más elaborados como una regresión lineal, para la prueba se opta por una combinación de las dos.

Decidir: Se activa una alarma interna de síntoma, en caso en que se detecta una tendencia al alza en el procesamiento de los núcleos del

sistema y se presenta una tendencia al alza en la utilización de memoria de intercambio.

Actuar: En este caso se genera un síntoma con identificador "*Degeneración del ambiente*" y mantenerlo como registro histórico.

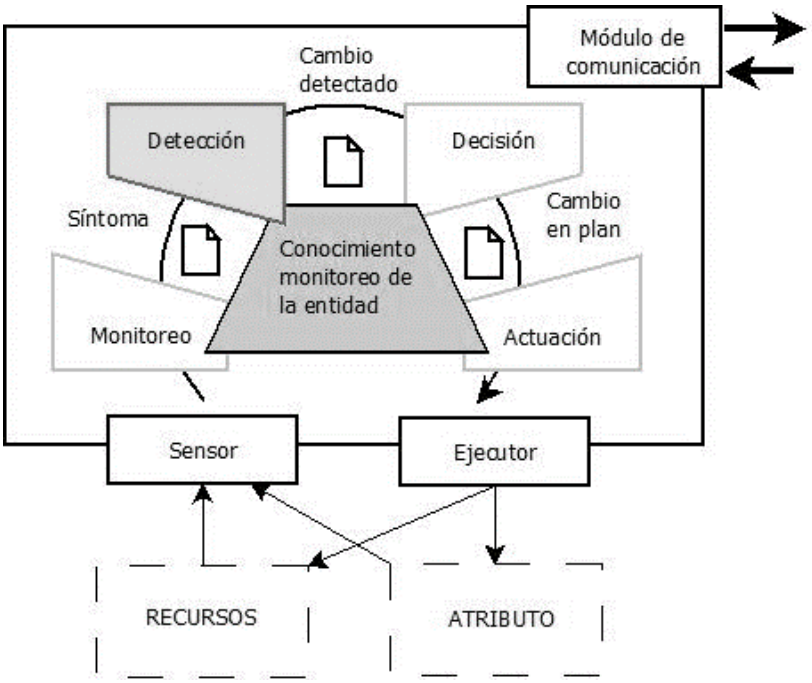
6.3 ETAPA DE DETECCIÓN

6.3.1 Diseño de agente decisión

El objetivo de los agentes de detección es proveer mecanismos para analizar *situaciones* y determinar si algún cambio necesita ser llevado a cabo. De nuevo, se aclara que no se dice que estas *situaciones* se consideran solo anómalas, por el contrario, si se reflexiona con cuidado es posible encontrar condiciones que permitan dar paso a oportunidades de mejora, todo depende de la combinación adecuada de síntomas.

Recuerde la estructura de un agente de detección:

Figura 34. Agente de detección



Fuente. Elaboración personal

El agente de detección se apoya de los síntomas identificados en el monitoreo, y a partir de analizar la ocurrencia de uno o varios de ellos, genera un aviso de cambio. Entre las restricciones tenidas en cuenta para el diseño de estos agentes se tiene: *la correlación de síntomas y eventos generados al detectar cambios, la influencia del módulo de conocimiento, el establecimiento de políticas o reglas.*

De la etapa de monitoreo, los agentes diseñados generan los síntomas:

- Incremento en el número de conexiones
- Bloqueo de hilos trabajadores
- Tiempos excesivos de atención de solicitudes
- Cargas de trabajo no equitativas entre hilos trabajadores
- Degeneración del ambiente

Como producto de la detección la idea es generar eventos de cambio que puedan ser tomados por otros agentes, como por ejemplo los agentes de decisión, para realizar sus tareas. Entre los cambios propuestos que se pueden generar del proceso de detección, a *groso modo* y con propósito de validación del marco de trabajo propuesto se generan:

- Sobrecarga inusual de trabajo
- Degradación de los tiempos de respuesta a las solicitudes
- Posibilidad cercana de un bloqueo del sistema

6.3.2 Correlación de síntomas y generación de eventos de cambio

A continuación, se debe analizar la relación entre los síntomas y los eventos de cambio para de esta manera facilitar la detección, la idea principal es identificar correlación de síntomas y su relación con los eventos a generar, teniendo en cuenta los parámetros previamente listados se pueden establecer las siguientes políticas de generación de eventos.

	Sobrecarga de trabajo
Incremento en el número de conexiones	*Si se alcanza o se está muy cerca de alcanzar el límite de conexiones máximo.*
Bloqueo de hilos trabajadores	*Revisar que no haya estado de bloqueo y que el número de hilos activos esté cercano al máximo.*
Tiempos excesivos de atención de requerimientos	*Si se presenta este síntoma y el número de colas e hilos es cercano al máximo.*
Cargas de trabajo no equitativas entre hilos trabajadores	*Revisar que no se esté presentando este síntoma.*

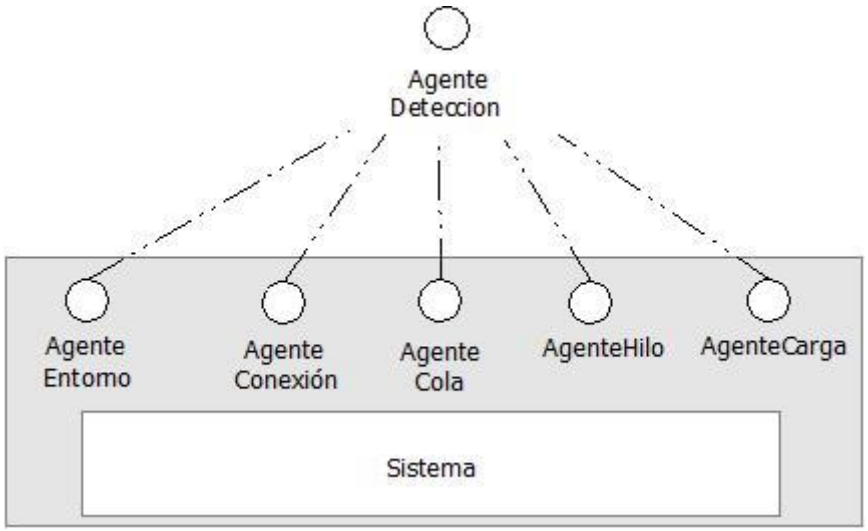
	Degradación de tiempos de respuesta
Incremento en el número de conexiones	*Este evento es indiferente a este tipo de síntoma.*
Bloqueo de hilos trabajadores	*Si se está presentando este síntoma. Y se está cerca del número máximo de hilos.*
Tiempos excesivos de atención de requerimientos	*Si se está presentando este síntoma y se está cerca al tamaño de colas de espera máximo y cerca al tiempo máximo de procesamiento por hilo.*
Cargas de trabajo no equitativas entre hilos trabajadores	*Si se presenta este síntoma independiente de cualquiera de los anteriores.*

	Posibilidad de bloqueo del sistema
Incremento en el número de conexiones	*Se presenta este síntoma y estamos cerca del máximo número de conexiones.*
Bloqueo de hilos trabajadores	*Se presenta este síntoma.*
Tiempos excesivos de atención de requerimientos	*Se presenta este síntoma.*
Cargas de trabajo no equitativas entre hilos trabajadores	*No importa el estado de este síntoma.*

6.3.3 Influencia del módulo de conocimiento

Un último aspecto que considerar para el diseño de los agentes de detección es la influencia del módulo de conocimiento, básicamente, algunas de las políticas podrían verse afectadas por datos históricos registrados en dicho módulo. Así, por ejemplo, si se ha registrado la ocurrencia de ciertos síntomas y ello ha derivado en la generación de un evento, la política podría cambiar a la identificación de dicho patrón. Para conservar un alcance factible dentro de esta sección se deja fuera de esta aproximación de diseño este punto, pero debería considerarse en caso de tener políticas dinámicas.

Figura 35. Continuación de la jerarquía hasta detección



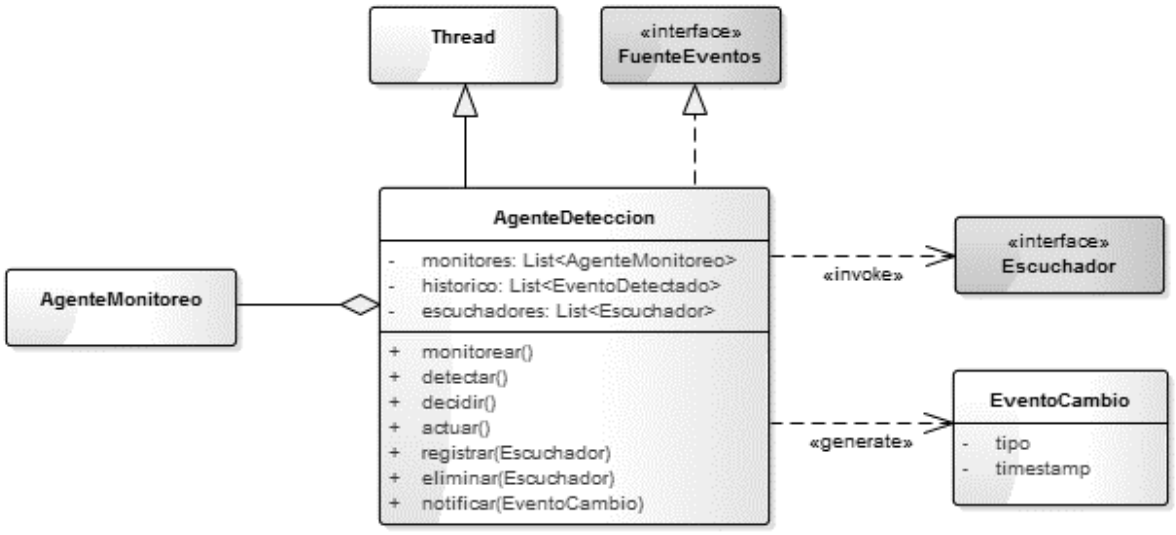
Fuente. Elaboración personal

La imagen de la Figura 36 muestra cómo el nuevo agente se organiza con respecto a los agentes de monitoreo, de nuevo se recuerda que esta organización se hace respetando la estructura jerárquica mencionada con anterioridad.

6.3.4 Diseño del agente

La propuesta de diseño técnico para el agente de detección se presenta en la imagen de la Figura 37.

Figura 36. Diseño de agentes de detección



Fuente. Elaboración personal

Anteriormente se había mencionado que este tipo de agentes seguiría un patrón de fuente de eventos para manejar su comunicación con los agentes de decisión, como se aprecia, las capacidades de dicho agente incluyen la manera de *registrar y eliminar* escuchadores de sus futuros eventos en este caso *EventoCambio* que contiene información relacionada al evento que se ha identificado, de esta forma la clase **AgenteDeteccion** puede *notificar* dicho evento a la lista de escuchadores registrados. Este agente tiene un listado de agentes monitor con los cuales intercambiará información sobre los síntomas cada cierto tiempo, para llevar a cabo las tareas del método *detectar* que emplea estos datos más los datos *históricos de eventos* para realizar la tarea de identificación de eventos de la manera más apropiada.

Dado que este es un agente especializado en la etapa de detección se explica cada método desde su objetivo primordial, el reconocimiento de eventos, además de los otros métodos mostrados en la figura.

AgenteDeteccion

Monitorear: Consiste en la lectura de síntomas identificados para cada uno de los elementos de la lista de agentes monitores.

Detectar: Implica el mapeo entre política de detección de cambios y el tipo de evento a generar (ver correlación entre síntomas y eventos de cambio).

Decidir: Se contrasta contra el histórico de eventos y parámetros del sistema y se decide si se genera o no un evento de cambio.

Actuar: Consistiría en la creación del evento de cambio, la notificación de este a los escuchadores y el almacenamiento en el histórico de eventos.

Registrar: Este método permite adicionar un nuevo agente escuchador a la lista de escuchadores, en particular todos estos agentes registrados recibirán notificaciones de eventos de cambio.

Eliminar: Permite desvincular un agente escuchador.

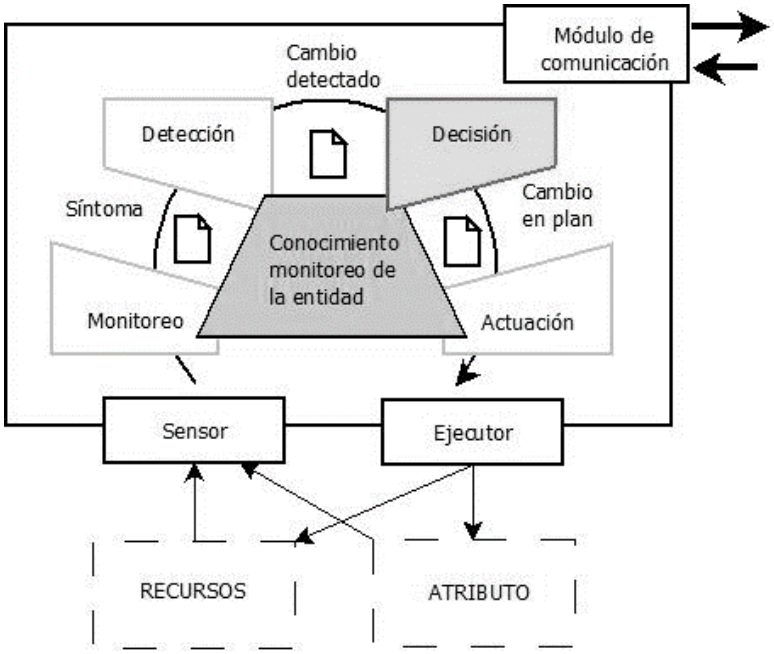
Notificar: Recorre cada elemento de la lista de escuchadores y les entrega una instancia compartida del evento identificado.

6.4 ETAPA DE DECISION

Luego de la generación de eventos, producto de la etapa de detección, los agentes de decisión seleccionan un procedimiento para promulgar una alteración al recurso administrado. Como se mencionó en el capítulo anterior esto puede tomar muchas formas, la propuesta para este caso de estudio es mirar hacia adelante hacia los recursos que podrían ser modificados. Y trazar hacia atrás los agentes especializados en esta etapa considerando las oportunidades de cambio.

Para comenzar la imagen de la Figura 38 presenta las entradas y salidas de la etapa que se enfatiza en dicho agente. Los agentes de decisión pueden llegar a realizar un análisis selectivo sobre la viabilidad de diferentes esquemas de cambio y la competencia entre ellos.

Figura 37. Agente de decisión



Fuente. Elaboración personal

Recuerde que el cambio corresponde a los eventos generados de la sección anterior, los cuales incluyen:

- Sobrecarga de trabajo
- Degradación de tiempos de respuesta
- Posibilidad de bloqueo del sistema

El *plan de cambio* se refiere a la selección de un esquema de cambio que da las pautas para los agentes de actuación de la siguiente etapa. Las consideraciones para el diseño del agente de decisión incluyen: esquemas en competencia, selección del agente de actuación, acciones del agente.

6.4.1 Esquemas en competencia

Los esquemas en competencia son el núcleo de este tipo de agentes, pueden darse como una lista de posibles cambios sugeridos para mitigar los efectos

negativos de un evento o para mejorar el estado actual del sistema. Sin embargo, se puede presentar múltiples esquemas para una misma situación, la manera de elegir entre u otro es manejar mecanismos de recompensa, premiando o castigando a los esquemas en función de su efectividad u otros factores de medición.

Para ilustrar este punto, considere el evento *sobrecarga de trabajo* y a continuación algunos esquemas de cambio que pueden proponerse para manejar la situación.

Esquema de cambio 1: *escalar en Y con detención de servicio*

- Detener instancia *Servidor* activo
- Crear réplica de la instancia *Servidor*
- Crear instancia de *distribuidor* de carga
- Conectar todas las instancias del *Servidor* al *distribuidor de carga*
- Iniciar instancias Servidor
- Iniciar distribuidor de carga

Esquema de cambio 2: *escalar en Y sin detención de servicio*

- Crear 2 réplicas de la instancia *Servidor*
- Crear instancia de *distribuidor* de carga
- Conectar ambas instancias del *Servidor* al *distribuidor de carga*
- Iniciar instancias *Servidor*
- Iniciar distribuidor de carga
- Poner distribuidor de carga a atender solicitudes
- Dejar que la carga del *Servidor activo* termine
- Finalizar *Servidor activo*

Esquema de cambio 3: modificar el parámetro *Número Máximo de Conexiones*

- Incrementar el parámetro número máximo de conexiones

- Incrementar en 1 el *número de hilos*

Como se puede observar, puntualmente son 3 esquemas de cambio completamente diferentes y cualquiera puede mejorar o no el estado del sistema, es por esta razón que es necesario que cada esquema se acompañe de un mecanismo de recompensa o castigo que permita en instancias del evento decantar hacia algún esquema particular, inicialmente cualquier elección entre ellos es igual de buena.

Ahora se considera el *evento degradación del tiempo de respuesta,* esto puede presentarse por diversas causas, como se presentó en la sección anterior (refiérase a ella para mayor detalle), algunos esquemas de cambio que pueden elegirse para manejar la situación.

Esquema de cambio 1: modificar el parámetro *tiempo de atención por petición* (esto podría balancear la carga de trabajo del sistema).

- Disminuir el parámetro tiempo máximo de atención por mensaje en un *quantum* de milisegundos.
- Reiniciar el *Timer* para actualización de tiempo de conteo para la petición

Esquema de cambio 2: disminuir el parámetro *el número máximo de hilos trabajador en el sistema* (esto puede incrementar la cantidad de procesamiento por hilo de trabajo y eliminar hilos bloqueados), anteriormente se dijo que la efectividad de estos esquemas depende de la estrategia de atención, para este ejemplo se utiliza un hilo por cola, con esto en mente.

- Seleccionar un hilo trabajador como candidato a eliminación
- Llevar el hilo a estado *SUSPENDIDO*
- Extraer el hilo de la lista de hilos del sistema
- Eliminar el hilo seleccionado
- Distribuir la carga en la cola de este hilo al resto de colas de espera

- Eliminar la cola de espera asociada
- Disminuir en uno el parámetro número máximo de hilos

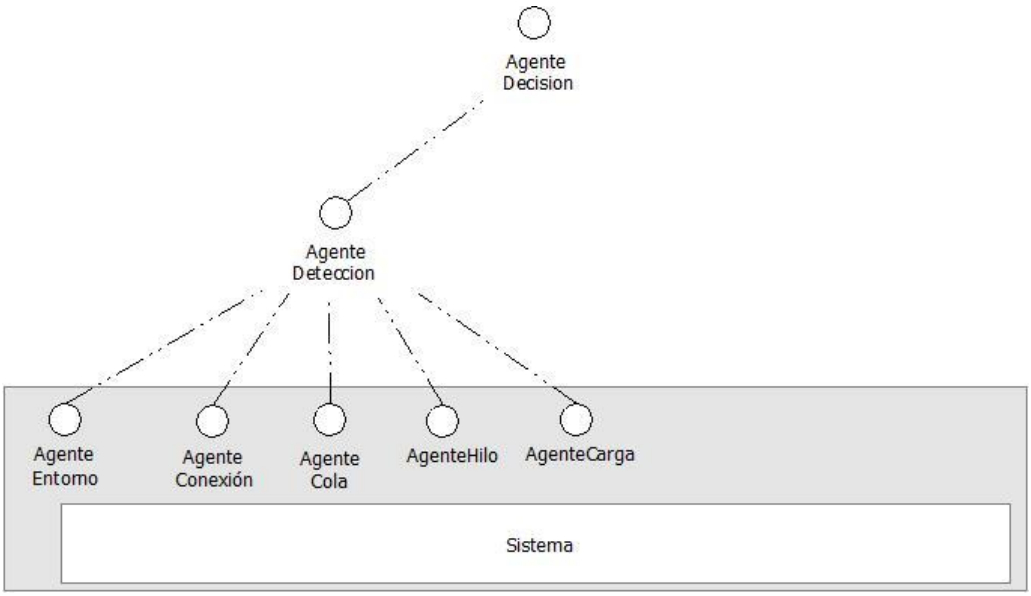
6.4.2 Selección del agente de actuación

El esquema de cambio seleccionado es implementado por un agente de actuación, en la práctica corresponde a aquel agente que posee la mejor puntuación, este debe ejecutar la lista de acciones mencionadas previamente. La manera en que se sugiere seleccionar un agente de actuación sobre los otros es realizar un análisis de la puntuación de cada agente, además, de considerar información propia de los elementos bajo custodia.

La idea de seleccionar uno de los agentes es que se realicen micro-ajustes de manera segura sin perturbar la operación del sistema.

La imagen de la Figura 39 muestra cómo el nuevo agente se organiza con respecto al resto de agentes del sistema, se insiste en que esta organización se hace respetando la estructura jerárquica de agentes. Como se aprecia, también para esta etapa se creará un único agente de decisión quien tiene la responsabilidad de elegir uno de los esquemas de cambio entre aquellos que tiene bajo su control, y evaluar la efectividad de este. Para el caso de estudio que nos incumbe sólo tenemos un único agente, esto debido a la decisión previa de controlar un único evento de los previamente detectados, sin embargo, es posible tener más agentes para controlar uno o más de estos eventos, la decisión depende del diseñador, los agentes de decisión son libres de registrarse a cualquier evento, de esta manera pueden ser informados de múltiples eventos, e incluso en su lógica de control manejar combinaciones lógicas entre ellos para inclinarse hacia alguno de los esquemas que maneje.

Figura 38. Jerarquía de agentes hasta decisión



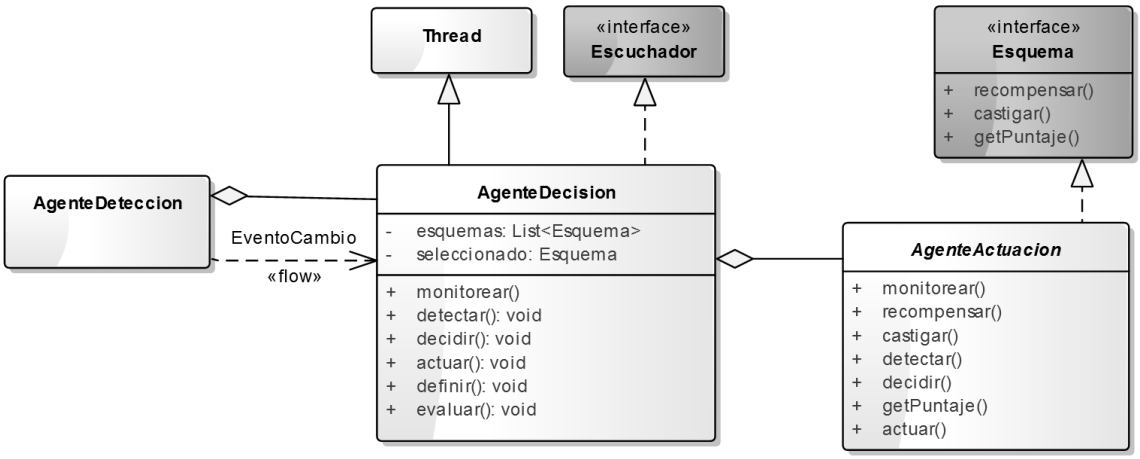
Fuente. Elaboración personal

6.4.3 Diseño de agente de decisión

La propuesta de diseño técnico para el agente de decisión se presenta en la imagen de la Figura 40. De este vale señalar varios aspectos:

- Los agentes de decisión se registran a los agentes de detección como escuchadores de eventos.
- Todo agente de decisión conserva una lista de esquemas, donde básicamente el esquema para esta aproximación inicial es un plan de cambios (como los expuestos al inicio de la sección).
- Cada esquema es implementado por un agente de actuación que será explicado en la siguiente sección que además ofrece una interfaz de evaluación para recompensa o castigo, y conserva el estado la misma en la puntuación.
- El agente de decisión tiene las tareas de seleccionar uno de los esquemas en competencia propios, y evaluar la efectividad y eficiencia de este.

Figura 39. Diseño del agente de decisión



Fuente. Elaboración personal

AgenteDecision

Monitorear: Consiste en la lectura de la información una vez se captura un evento de cambio.

Detectar: Preparación de los posibles esquemas de cambio a ser tenidos en consideración.

Decidir: En función de los valores de la evaluación realizada a los diferentes esquemas se escoge alguno de los esquemas considerados.

Actuar: Consistiría en establecer en la pizarra en nuevo esquema elegido para su aplicación por parte de los agentes de actuación.

Evaluar: Permite valorar la efectividad del esquema seleccionado a partir de información sobre eficiencia del cambio aplicado.

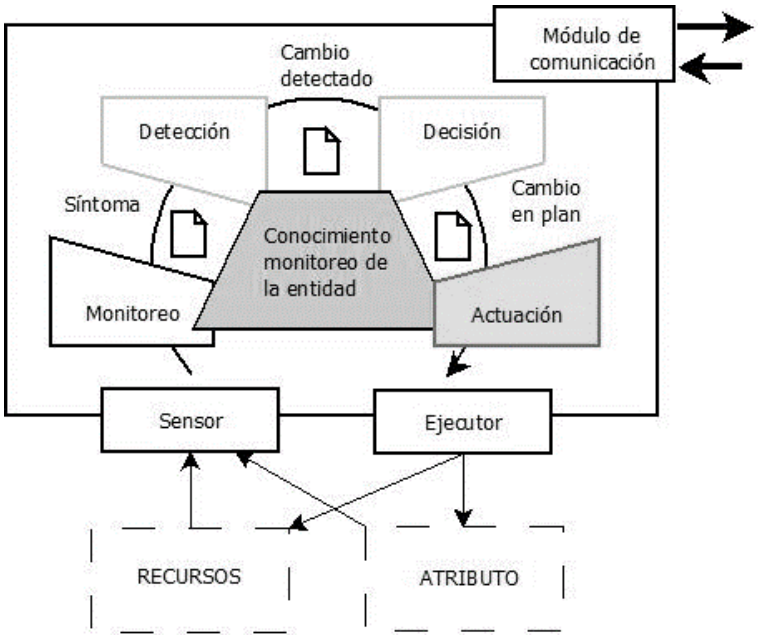
6.5 ETAPA DE ACTUACIÓN

Una vez elegido el *esquema de cambio* o *agente de actuación*, éste iniciará su tarea de modificar un aspecto particular del sistema, bien sea en su estructura o comportamiento realizando las acciones necesarias para mantener operativo

el mismo. Entre las consideraciones para tener en cuenta para el diseño de estos agentes están: *oportunidades del cambio, estrategia y la táctica, propagación del cambio y continuidad de la operación.*

Cuando el agente de actuación seleccionado es iniciado, se da a la tarea de revisar el estado actual del elemento sobre el cual controla el cambio, la idea principal de esta revisión es censar el elemento bajo su dominio y determinar la mejor manera de desplegar el cambio sugerido para pasar a las acciones que lo realizan. La imagen de la Figura 41 presenta las entradas y salidas de la etapa que se enfatiza en dicho agente. Como se observa con claridad, la entrada es el *plan de cambio* definido en la implementación de las acciones del agente de actuación seleccionado quien finalmente activa los mecanismos de cambio.

Figura 40. Agente de actuación



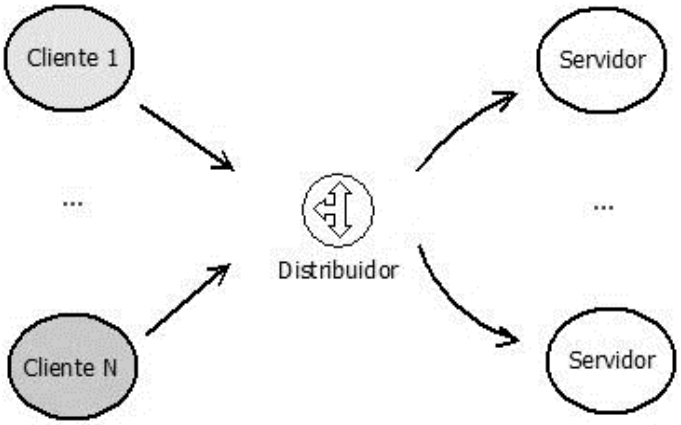
Fuente. Elaboración personal

6.5.1 Oportunidades de cambio

Esta sección explica algunas oportunidades de cambio del diseño del aplicativo en estudio:

- Utilizar el *escalar en Y*, creando réplicas completas de la instancia del elemento *Servidor y* anteponer un *distribuidor de carga* previo a todas las instancias de estos. Se tiene la ventaja de explotar las capacidades de planeación de procesos del sistema operativo, además de permitir la distribución en múltiples nodos, no necesariamente geográficamente juntos.

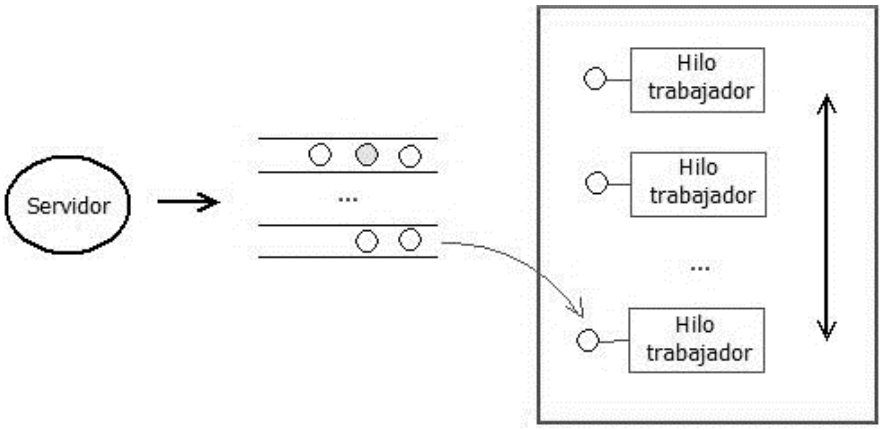
Figura 41. Escalar la aplicación en el eje Y



Fuente. Elaboración personal

- Incrementar/disminuir el número de instancias de *Hilo trabajador,* los hilos de trabajo son quienes finalmente realizan los cálculos, en ocasiones modificar la cantidad de ellos puede beneficiar los atributos de calidad del aplicativo, también como impactarlos negativamente.

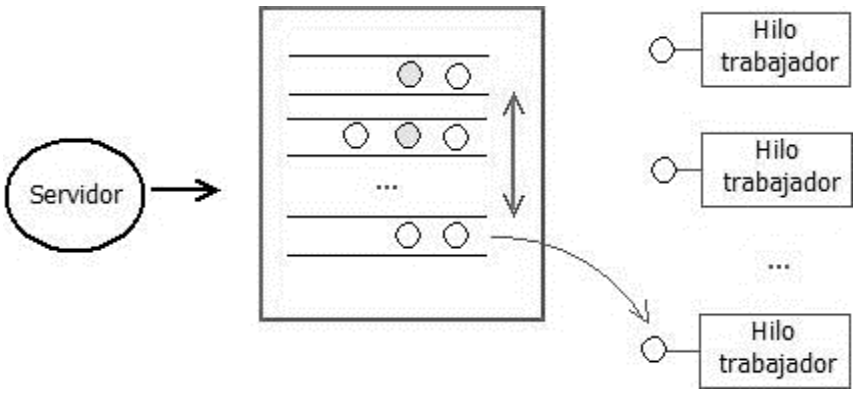
Figura 42. Incremento/Disminución de Hilos trabajadores.



Fuente. Elaboración personal

- Incrementar/disminuir el número de colas de espera, esta medida bien es un cambio factible, sin embargo, su impacto real depende de la estrategia de atención utilizada, pues para el caso de *hilos x cola*, se crea un nuevo *hilo trabajador* dedicado a la cola creada.

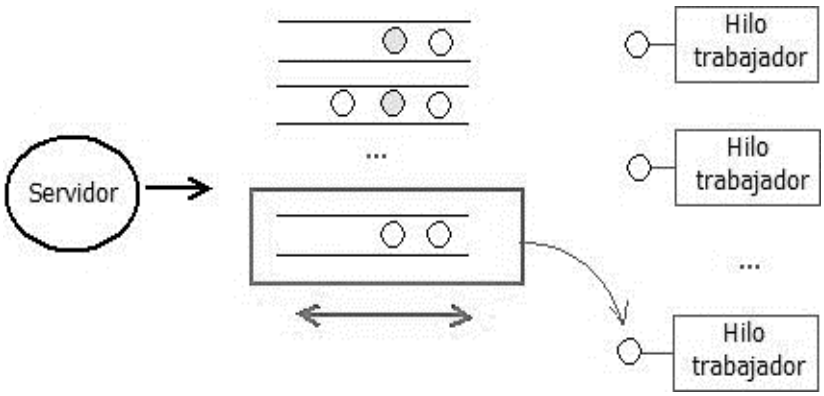
Figura 43. Incremento/Disminución de colas de espera



Fuente. Elaboración personal

- Variar el tamaño de cada cola de espera, igual que en el caso anterior su impacto y efectividad depende de la estrategia de atención, por ejemplo, para hilo x cola, uno de los hilos puede llegar a tener más carga que el resto.

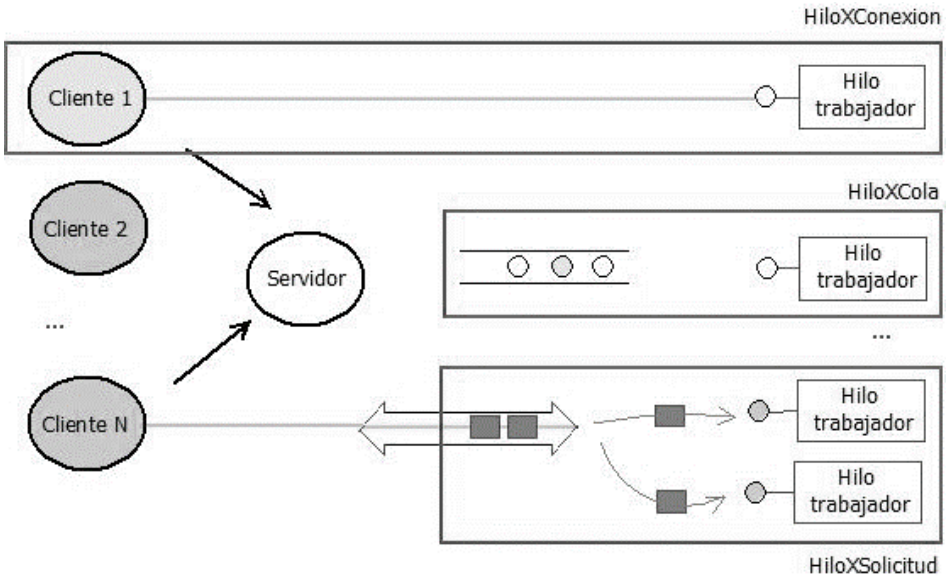
Figura 44. Cambio del tamaño de cola de espera



Fuente. Elaboración personal

- Cambiar la estrategia de atención (HiloXConexion, HiloXCola, HiloXMensaje), la estrategia de atención tiene que ver con la manera en que se organizan los hilos respecto de los elementos que componen el aplicativo, por ejemplo, en la estrategia de *hilo x conexión* se tiene un hilo dedicado a una conexión establecida por un cliente, este hilo atenderá y operará todos los mensajes recibidos por dicha conexión. En la estrategia de *hilo x cola* la dedicación de un *hilo trabajador* se realiza una cola de espera, de esta manera dicho hilo atenderá todas las conexiones que sean puestas en esta cola de espera. Hilo x mensaje, implica que el *hilo trabajador* que recibe una *petición de operación* toma el mensaje y lo entrega a una copia suya para realizar la operación mientras el hilo trabajador original sigue atendiendo otros mensajes. La estrategia de atención puede llegar a ser el tipo de cambio que más afecte los atributos de calidad de la aplicación.

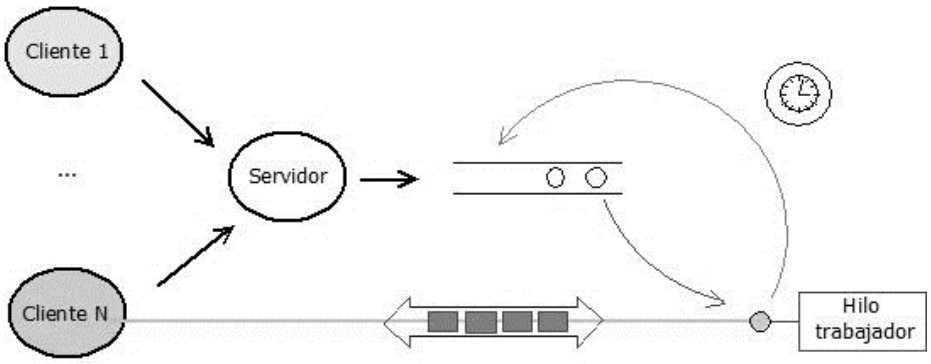
Figura 45. Cambio de estrategia de atención



Fuente. Elaboración personal

- Modificar el tiempo de dedicación a una tarea, volviéndola a poner en cola en caso de agotarse. Esto es tener un temporizador que le indique a un hilo trabajador encolar de nuevo el cliente atendido y atender a otro de la cola de espera.

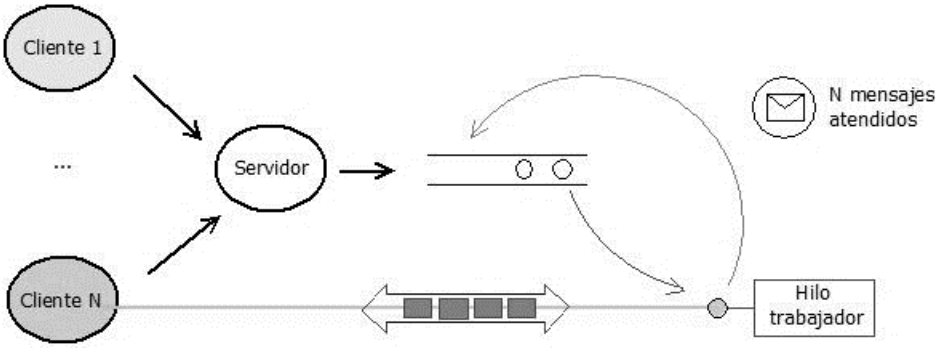
Figura 46. Cambio en tiempo de dedicación



Fuente. Elaboración personal

- Atender sólo un número de mensajes por conexión, encolar la conexión. Similar al anterior sólo que en lugar del tiempo se toma en cuenta el número de mensajes atendidos, cada cierto valor se cambia de cliente.

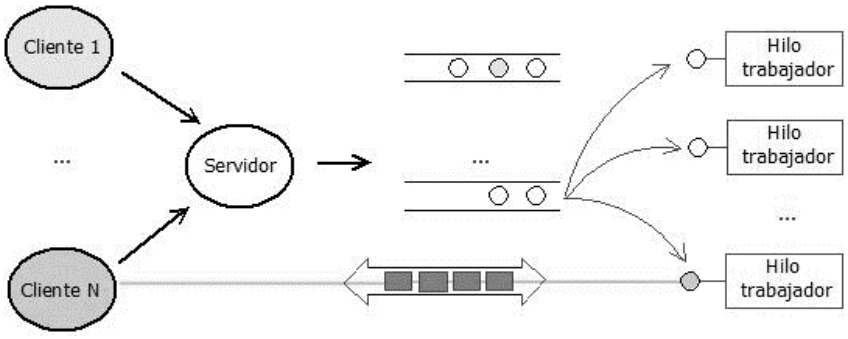
Figura 47. Retornar petición a cola de espera



Fuente. Elaboración personal

- Asignar a una cola más *hilos de trabajo*. Esto es que los hilos tomen una conexión y su carga de trabajo asociada de la misma cola de espera.

Figura 48. Dedicar hilos a colas específicas

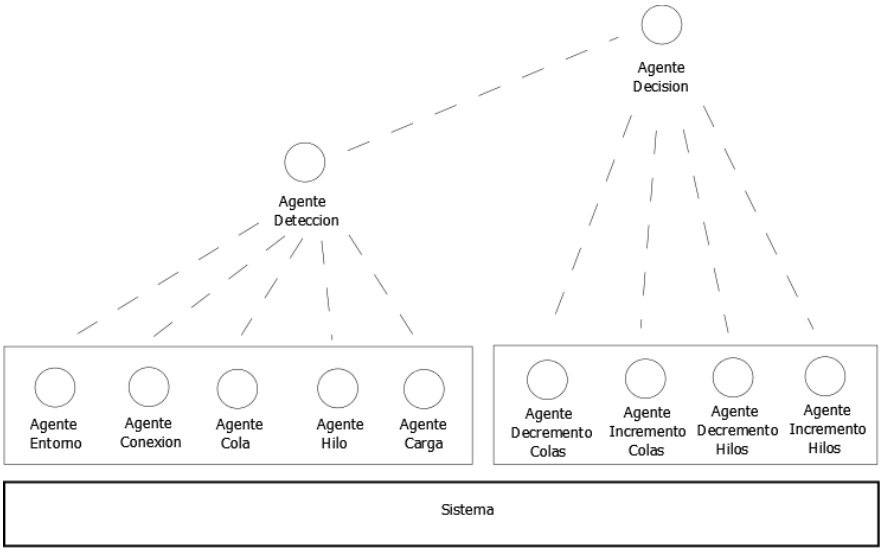


Fuente. Elaboración personal

6.5.2 Estrategia y táctica

La imagen de la Figura 50 muestra cómo los nuevos agentes se organizan con respecto al resto de agentes del sistema, esta organización respeta la estructura jerárquica de agentes definida y, de hecho, para este ejemplo los agentes se conectan como hijos del agente de decisión, pero esto no obedece a ninguna regla, en cierta forma la única restricción es mantener una estructura jerárquica con la intención de posteriormente explotar las ventajas de esta.

Figura 49. Jerarquía hasta elementos de actuación



Fuente. Elaboración personal

Dado que *la estrategia* responde a la pregunta: "*¿qué se debe hacer?*", ésta hace referencia al *plan de cambio* o agente de actuación seleccionado. Por su parte, *la táctica* responde a la pregunta: "*¿cómo se debe hacer?",* en este caso, *¿cómo llevar a cabo el cambio particular seleccionado*? Para dar respuesta a esta cuestión se considera que cada agente de actuación debe tener en cuenta los siguientes aspectos:

1) ¿Cuál es el estado actual del recurso a cambiar?, es decir, se encuentra activo o inactivo, queriendo significar con ello si el recurso está siendo utilizado por la ejecución de una tarea o si por el contrario se encuentra libre.

2) ¿Cuáles son los pasos por seguir para desplegar el cambio sin afectar la operación del resto de los elementos?

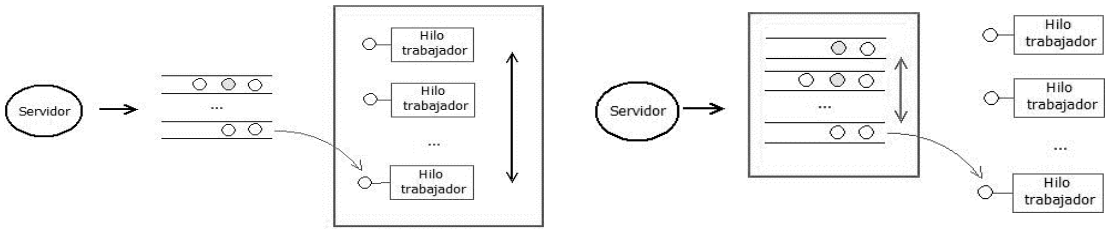
Las estrategias seleccionadas para adaptación teniendo en cuenta las oportunidades de cambio y el caso de estudio son:

Estrategia 1

Incrementar/disminuir el número de instancias de *Hilo trabajador*.

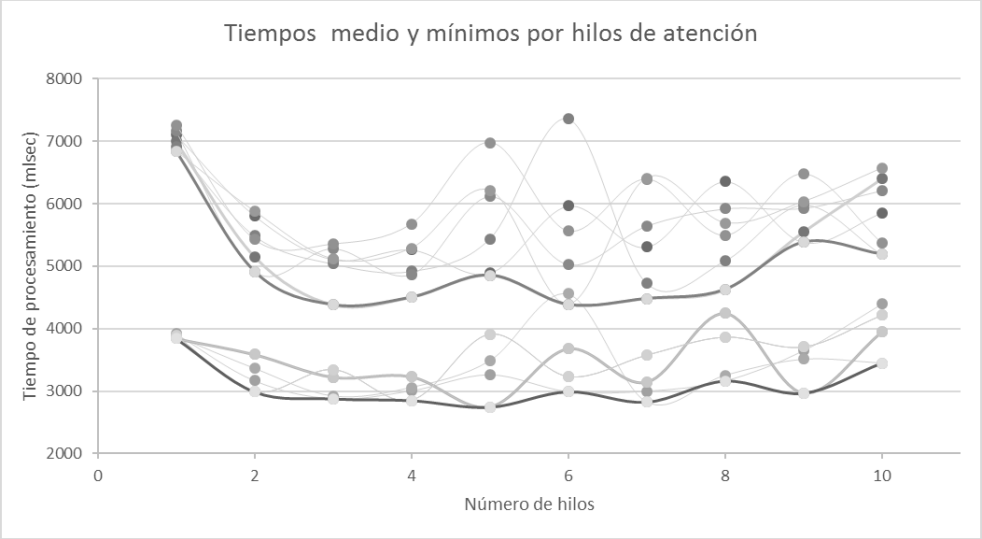
Estrategia 2

Incrementar/disminuir el número de colas de espera.



Tener varios hilos o colas de espera pueden mejorar el tiempo de promedio de respuesta de esta aplicación, pero se debe ser cuidadoso al tener muchas instancias de estos elementos ya que pueden comenzar a interferir con otros e impactar negativamente el rendimiento de la aplicación, la Figura 51 refleja precisamente esta situación.

Figura 50. Tiempos medios y mínimos por hilos de trabajo



Fuente. Elaboración personal

Se construyó un pequeño programa que permite variar el número de instancias del *pool* tanto para colas como para hilos, como se observa en la imagen para un escenario donde se instanciaron 100 clientes y un único servidor las diferencias en el tiempo promedio de respuesta por conexión varia significativamente, incluso se alcanzaron tiempos mínimos en combinaciones específicas de numero de colas y numero de hilos que redujeron el tiempo medio de respuesta a la mitad en comparación a un escenario con una única cola y un único hilo trabajador, es por esta razón que se decide habilitar el componente *Servidor* con actuadores que permitan microajustes sobre estos dos elementos de la arquitectura del programa, y por consiguiente las estrategias seleccionadas.

Si el agente de actuación seleccionado es aquel con el plan de cambio: "*decrementar el número de colas"* el agente de actuación es el responsable de implementar la táctica, y puede hacerlo de la siguiente forma:

1) Se valida la existencia de instancias de Colas y se verifica que la cantidad de ellas es mayor que 1 (uno). Se revisa uno a uno el estado de cada cola para determinar cuál de ellas seleccionar para su

eliminación, se comprueba la cantidad de mensajes pendientes de ser atendidos que contiene. En función de esta información se puede utilizar varios criterios de selección: uno de ellos puede ser seleccionar la cola con menor número de mensajes, o aproximar la complejidad subyacente del procesamiento del mensaje, a esto es lo que se hace referencia con revisar el estado del recurso. Una vez seleccionada la cola a eliminar se continua con el siguiente paso.

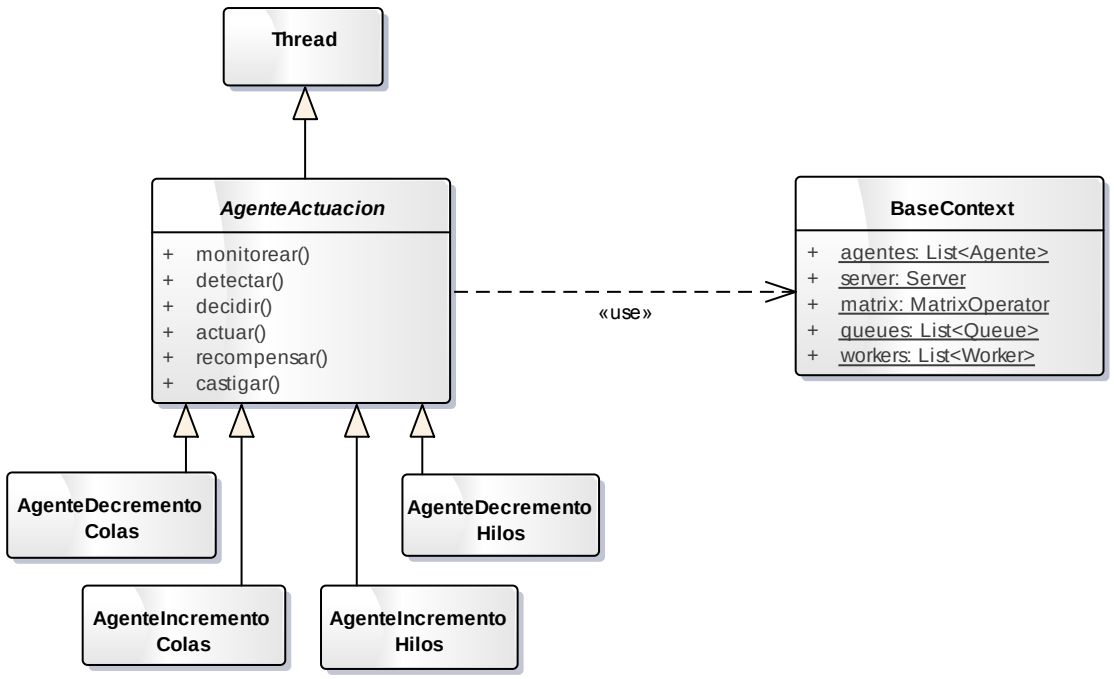
2) Una vez se seleccione la cola a eliminar esta se debe inactivar, esto quiere decir que se cambia su estado para no aceptar más mensajes. El hilo de actuación espera a que la cola seleccionada se haya vaciado por completo y una vez ocurra esto se elimina de la lista de colas, sin el temor de afectar la continuidad de la operación del sistema.

6.5.3 Diseño de agente de actuación

Una aproximación al diseño de un agente de actuación se expone en la imagen de la Figura 52. La clase *BaseContexto* permite tener acceso directo a los recursos o partes del sistema que pueden estar sujetos a las acciones de cambio, y así poder acceder a los mismos desde cualquier clase de actuación derivada, cada clase de actuación, tal cual fue mencionado, se encarga de la modificación de un aspecto en alguno de estos recursos. Los aspectos elegidos para el caso de estudio serian: incremento y decremento de colas, e incremento y decremento de hilos. En el capítulo de validación se da una explicación razonable del porque se han elegido estos dos aspectos para reflejar cambios.

Por simplicidad para el ejemplo en cuestión se opta por una estrategia que interrumpe la operación de algunos elementos mientras se lleva a cabo las operaciones de cambio.

Figura 51. Diseño del agente de actuación



Fuente. Elaboración personal

Se puede dar una aproximación a las acciones que llevaría a cabo un agente de actuación, aunque es importante mencionar que cada oportunidad de cambio puede requerir un agente particular que implemente las acciones necesarias para coordinar las acciones de cambio sobre el recurso o atributo custodiado.

AgenteActuacion

Monitorear: Consiste en la lectura de la información del recurso principalmente su estado.

Detectar: Definir que parte de la estructura o del comportamiento será objeto de la modificación.

Decidir: Analiza las condiciones limite establecidas para los recursos y el estado del mismo.

Actuar: Coordinar una tras otra las acciones para realizar el tipo de cambio sobre el recurso en custodia.

Recompensar: Permite puntuar o valorar positivamente la efectividad del esquema aplicado en función de las metas establecidas para el sistema.

Castigar: Permite puntuar o valorar negativamente la efectividad del esquema aplicado en función de las metas establecidas para el sistema.

6.6 CONCLUSION

- En este capítulo se mostró como aplicar los conceptos del marco de trabajo propuesto a un ejemplo de aplicación para introducir la adaptación, manteniendo una traza con los principios planteados en el capítulo anterior. El siguiente capítulo aplica la ficha de cambio diseñada en el capítulo 1 para evaluar la efectividad del marco de trabajo sobre un prototipo del caso de estudio desarrollado, los resultados de estas pruebas se recopilan y analizan para mostrar la efectividad (pros y contras) de la propuesta.

7. EVALUACIÓN DE LA PROPUESTA

7.1 METODOLOGÍA

En el primer capítulo fue mencionada una aproximación a la manera que se realizaría la validación del prototipo de marco de trabajo, allí se mencionaron varios elementos entre ellos: variables del diseño, instrumento de evaluación y finalmente el análisis de los resultados. En esta primera parte se comentará un poco más en detalle de qué forma se ha llevado a cabo la validación, y para ello iniciamos mencionando las preguntas que motivaron el estudio anterior:

7.1.1 Pregunta principal

¿Es posible adaptar el diseño de un sistema software distribuido, sin perturbar su funcionalidad, mediante la aplicación de un conjunto de reglas, principios y conocimientos basadas en el estudio de los sistemas adaptativos complejos?

7.1.2 Preguntas secundarias

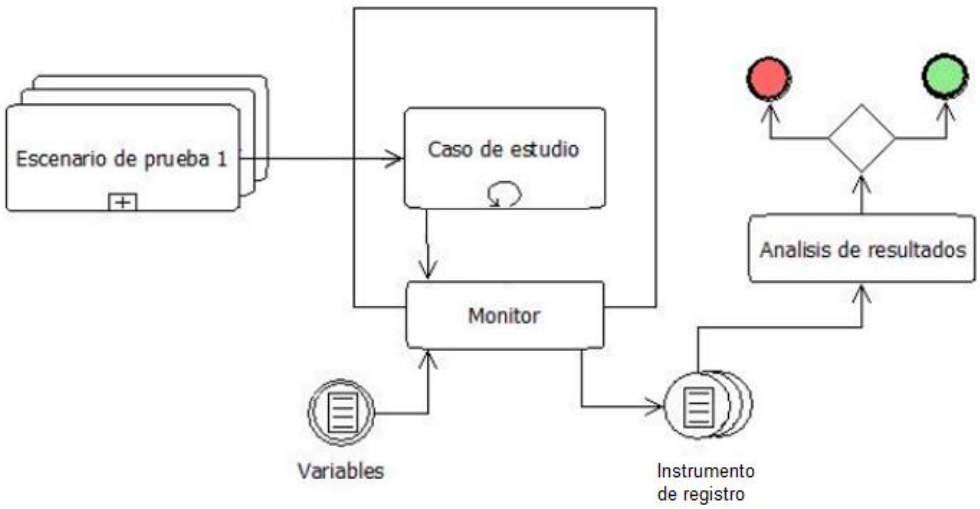
- En un diseño adaptable es posible efectuar cambios en términos de entidades que asumen roles específicos en el proceso.
- Las condiciones iniciales del sistema sesgan y limitan los mecanismos de ejecución de la adaptación y es uno de los factores más influyentes al dirigir el sistema hacia la misma.
- Se pueden realizar adaptaciones en tiempo de ejecución sin impactar negativamente el rendimiento del sistema.

7.1.3 Proceso de validación

Para validar el marco de trabajo se ha propuesto el siguiente proceso. Este proceso ya se había explicado en un capítulo previo, pero se darán detalles de este a continuación.

Para comenzar, se ha construido la aplicación Cliente/Servidor con la funcionalidad descrita previamente, se ha implementado en el lenguaje de programación Java que utiliza el paradigma de programación orientado a objetos, acorde con el diseño especificado. También, se implementa el marco de trabajo de forma tal que pueda ser activado o desactivado mediante registros de configuración, estos también poseen un set de propiedades para variar aspectos en el comportamiento del marco de trabajo que se describirán en breve.

Figura 52. Elemento de control



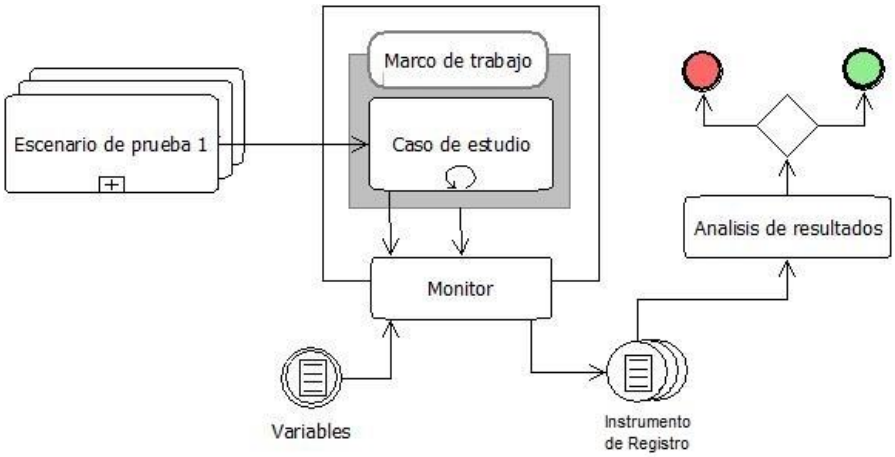
Fuente. Elaboración personal.

Como elemento de control el programa permite su ejecución sin marco de trabajo, conteniendo solo la lógica propia de la aplicación, sin incluir ningún elemento para habilitar la adaptación, como lo presenta la Figura 53. Básicamente, este será ejecutado para los mismos escenarios de prueba, a los cuales se harán mediciones de tiempos de respuesta. Dado que la adaptación se da cuando alguno de los *agentes de actuación* entra a realizar su función y modifica alguno de los elementos en el sistema, el estado s_i del programa es básicamente una combinación (*número de hilos, número de colas*) y la

adaptación se puede considerar básicamente como la transición de un estado inicial s_0 a un estado final s_f, denotado como ($s_0 \rightarrow s_f$), pensando en esto se han adicionado un par de propiedades de configuración que permiten compilar y ejecutar el programa para cualquier estado.

Luegos los mismos escenarios serán ejecutados pero esta vez para el caso de estudio con la aplicación del marco de trabajo y se realizará un proceso similar, como se muestra en la Figura 54. Así, se registran los tiempos para ambos estados (s_0, s_f), y luego se realiza la misma tarea, pero activando el marco de trabajo, se pretende validar la transición entre ambos estados y anotar los tiempos del programa. Con lo anterior se tendrá información suficiente para dar respuesta a las preguntas formuladas.

Figura 53. Prueba de escenarios con el marco de trabajo



Fuente. Elaboración personal.

Resumiendo lo anterior en una secuencia de pasos lógica y ordenada se tiene el siguiente procedimiento para la evaluación:

1. Construir la aplicación Cliente/Servidor con la funcionalidad descrita, permitiendo activar y desactivar el marco de trabajo.

2. Prediseñar una colección de escenarios de prueba para generar situaciones externas y condiciones internas que puedan desencadenar acciones de adaptación.
3. Utilizar como elemento de control los escenarios sobre el aplicativo con el marco de trabajo inactivo, para las configuraciones de elementos internas inicial y final, como se muestra en la Figura 53.
4. Probar los escenarios sobre el aplicativo del caso de estudio con el marco de trabajo activo (Figura 54).
5. Monitorear y registrar información de la ejecución del aplicativo. Al ejecutar cada escenario sobre el caso de estudio, un elemento de monitoreo (*que puede reducirse a un simple registro por salida estándar*) extrae información de la ejecución.
6. Analizar los resultados para evaluar si la situación de cambio es reconocida por el marco de trabajo y ésta desencadena una adaptación.
7. El análisis de resultados evalúa el impacto que tiene la adaptación en el desempeño obtenido de ambos programas.
8. Evaluar el impacto que tiene la adaptación en el desempeño realizando una comparación directa entre los resultados medidos del mismo escenario de prueba para las tres configuraciones de la aplicación:
    a. Inicial sin marco de trabajo
    b. Final sin marco de trabajo
    c. Transición entre ambos estados con marco de trabajo

7.1.4 Instrumento de validación

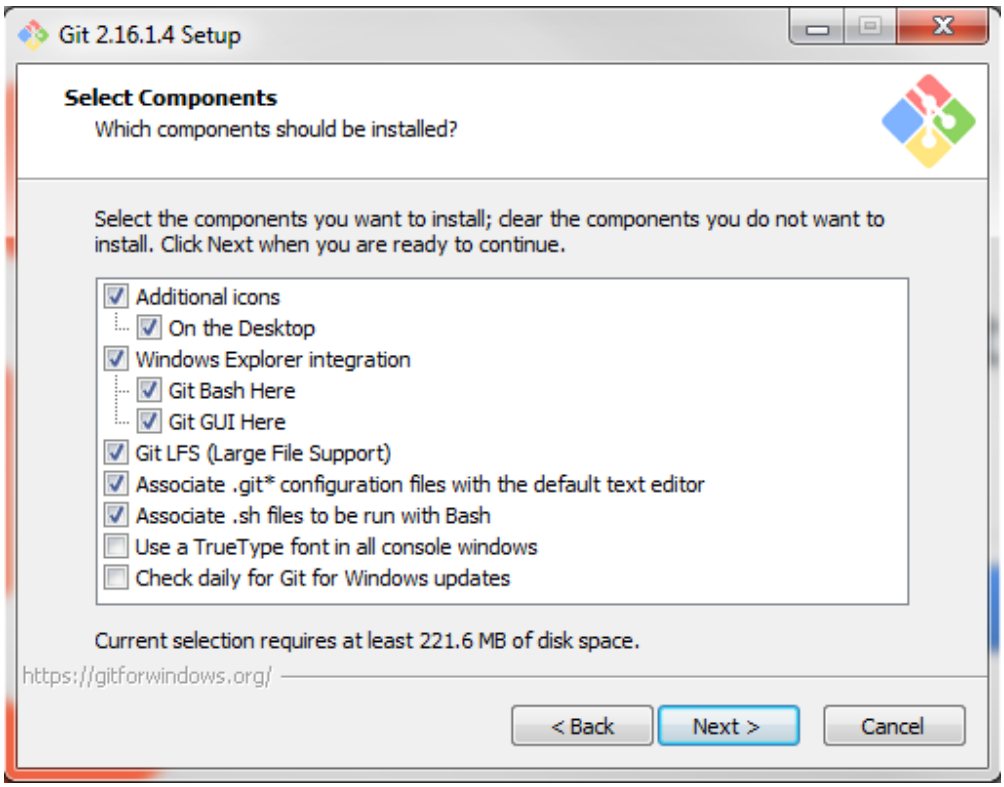
Entre las ideas detrás del diseño del instrumento de validación se tienen: está orientada por *escenarios de prueba*, el objetivo es determinar el grado de adaptación en diferentes escenarios para permitir contrastar los resultados obtenidos contra la hipótesis del trabajo y así validar la metodología de diseño para el marco de trabajo propuesto y los resultados y/o conclusiones.

7.1.6 Despliegue del proyecto

El objetivo de esta corta sección es indicar los pasos a seguir si se quiere ver el código fuente y ejecutar el mismo, el desarrollo y pruebas se realizaron sobre Windows 7 utilizando el IDE de desarrollo Netbeans.

Paso 1: Instalar ***Git Bash*** o una herramienta de línea de comandos para facilitar el trabajo con el repositorio del proyecto.

1. Descargar *Git SCM* de la url: https://gitforwindows.org/
2. Seleccionar el componente Git Bash una vez se inicia la instalación



3. Continuar con la instalación con las opciones que mejor considere para trabajar en su sistema

Paso 2: Instalar ***Java*** para su computadora de escritorio. El ***JRE*** es el Java Runtime Environment, en español es el Entorno de Ejecución de Java, en

palabras del propio portal de Java es la implementación de la Máquina virtual de Java que realmente ejecuta los programas de Java. El ***JDK*** es el Java Development Kit, que traducido al español es, Herramientas de desarrollo para Java, aquí nos encontraremos con el compilador javac que es el encargado de convertir nuestro código fuente (.java) en bytecode (.class), el cual posteriormente será interpretado y ejecutado con la ***JVM***.

1. Descargar ***JRE*** de la url: https://www.java.com/es/download/
2. Para modificar el código fuente se recomienda utilizar el IDE (Entorno integrado de desarrollo) NetBeans el cual, al descargarlo, también instala el ***JDK*** y puede hacerse desde el siguiente enlace: http://www.oracle.com/technetwork/es/java/javase/downloads/jdk-netbeans-jsp-3413139-esa.html

Paso 3: Descargar el código fuente de la aplicación.

1. Crear una carpeta en el sistema de archivos donde se quiera descarga el código fuente.
2. Dar click derecho sobre la carpeta y ejecutar la consola de comandos *Git Bash Here*
3. Conectar con los repositorios para descargar los códigos fuente de la aplicación servidor y la aplicación cliente:

   $ `git clone https://isokmio@bitbucket.org/isokmio/master-server-full.git`

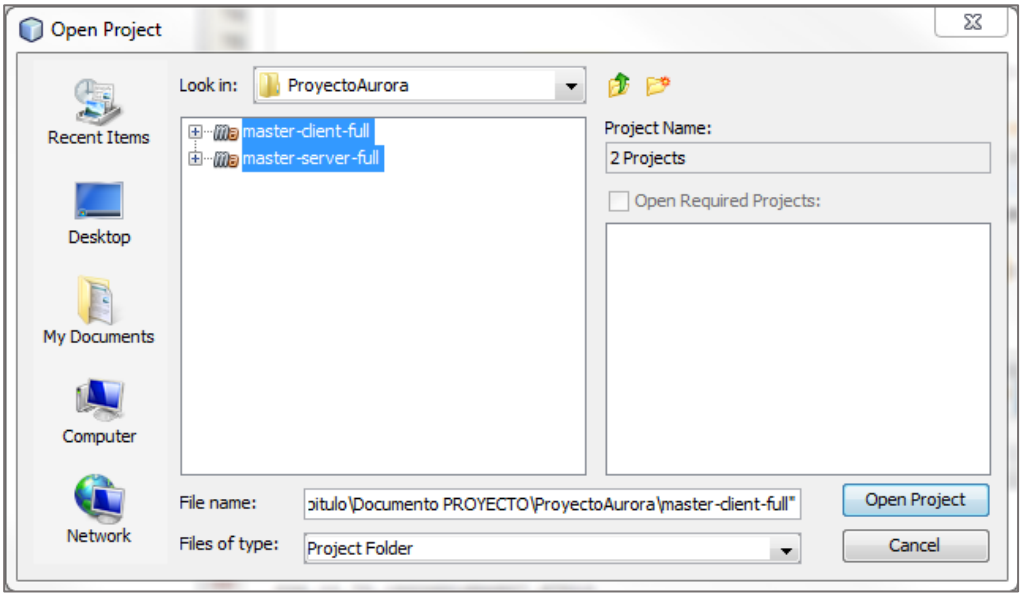
   $ `git clone https://isokmio@bitbucket.org/isokmio/master-client-full.git`

4. Lo anterior descarga dos proyectos con los nombres: master-client-full y master-server-full. Cada uno en su interior contiene el código fuente de la aplicación.

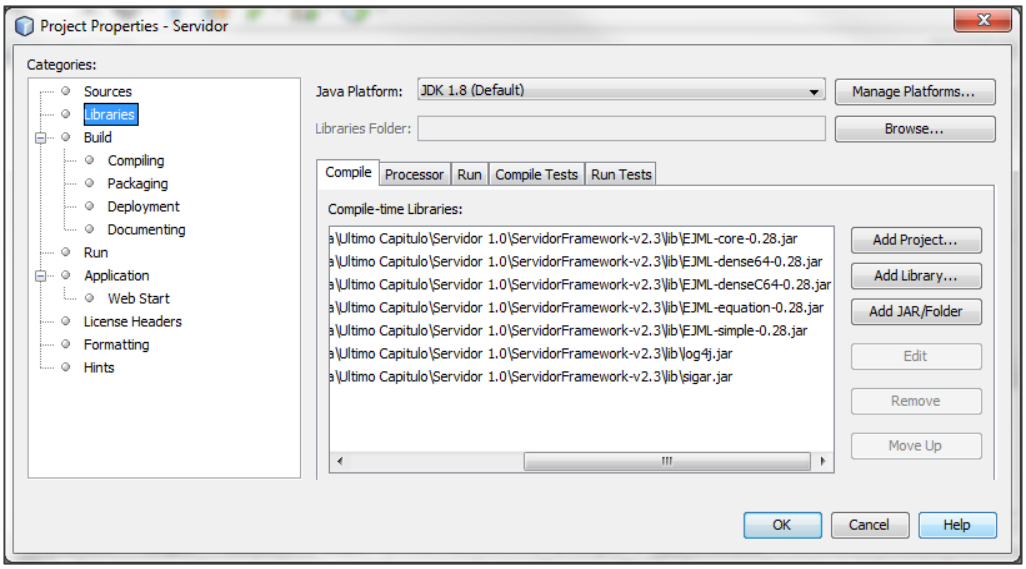
Paso 4: Abrir los proyectos con el IDE recomendado Netbeans.

1. Abril el Netbeans editor integrado de desarrollo.

2. Seleccionar *File>Open Project*
3. Seleccionar ambos proyectos y presionar *Open Project*.

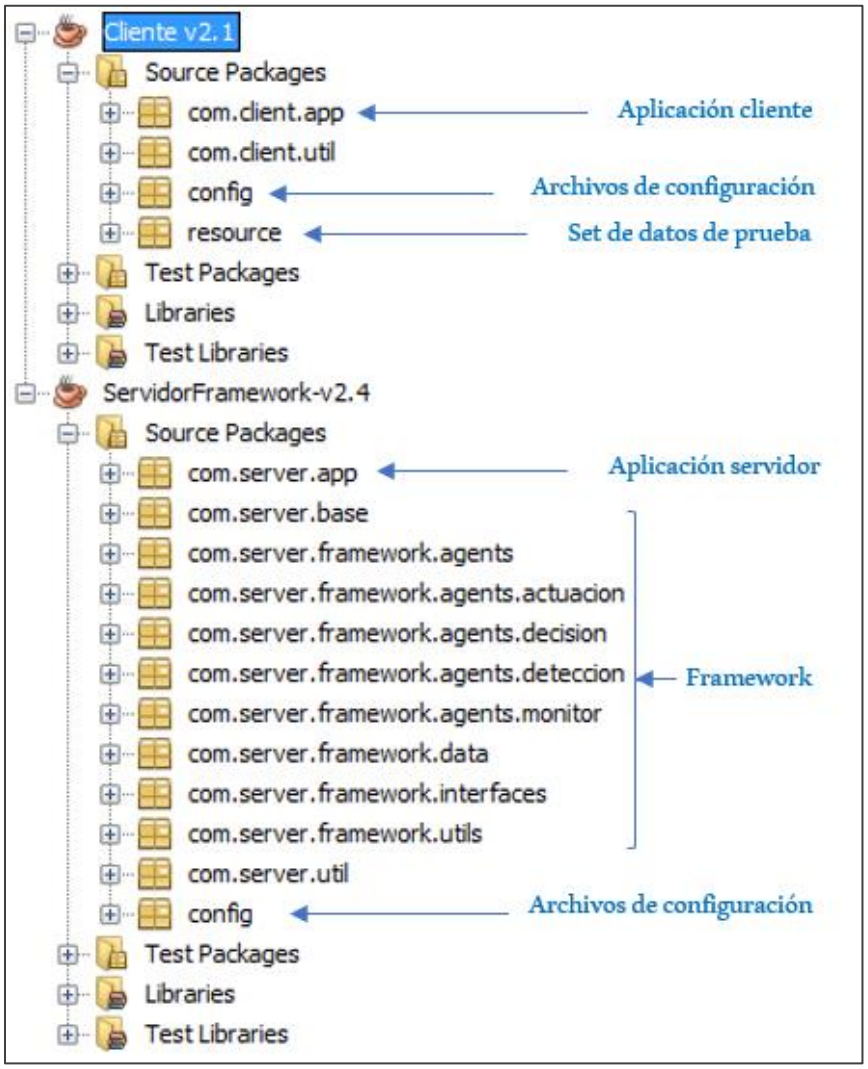


4. Las dependencias de librerías empleadas se resuelven mediante el sistema de gestión de dependencias Maven, por lo que puede ser posible que se requiera esperar mientras se cargan estas dependencias.

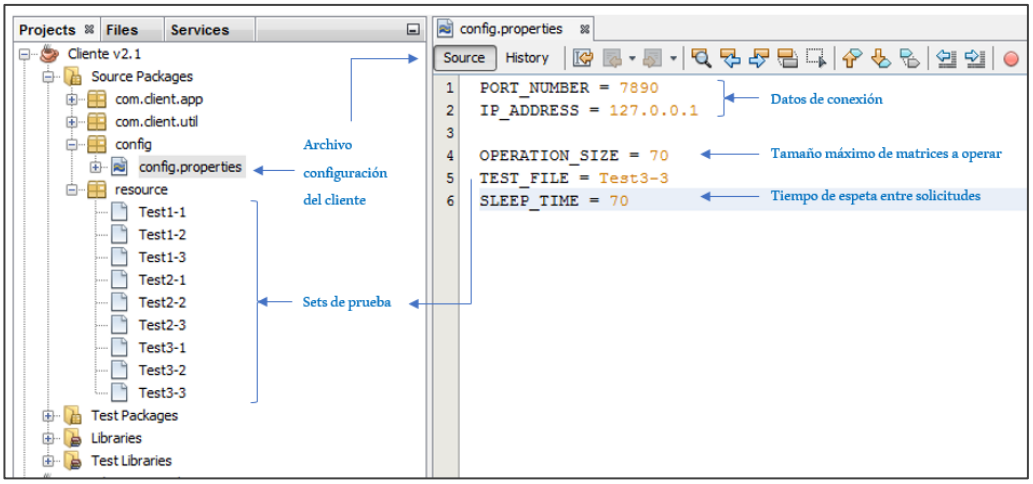


Paso 5: Estructura del proyecto.

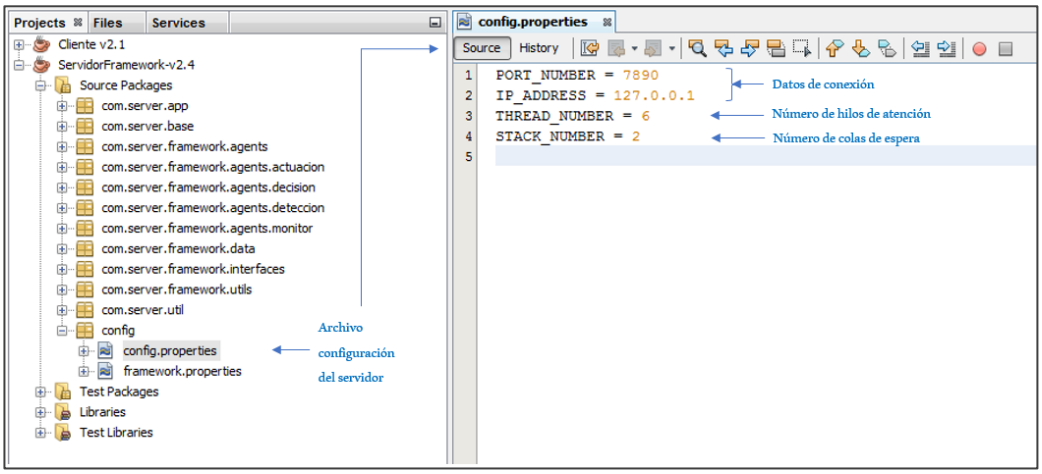
1. La estructura de proyectos y paquetes del desarrollo son las siguientes.



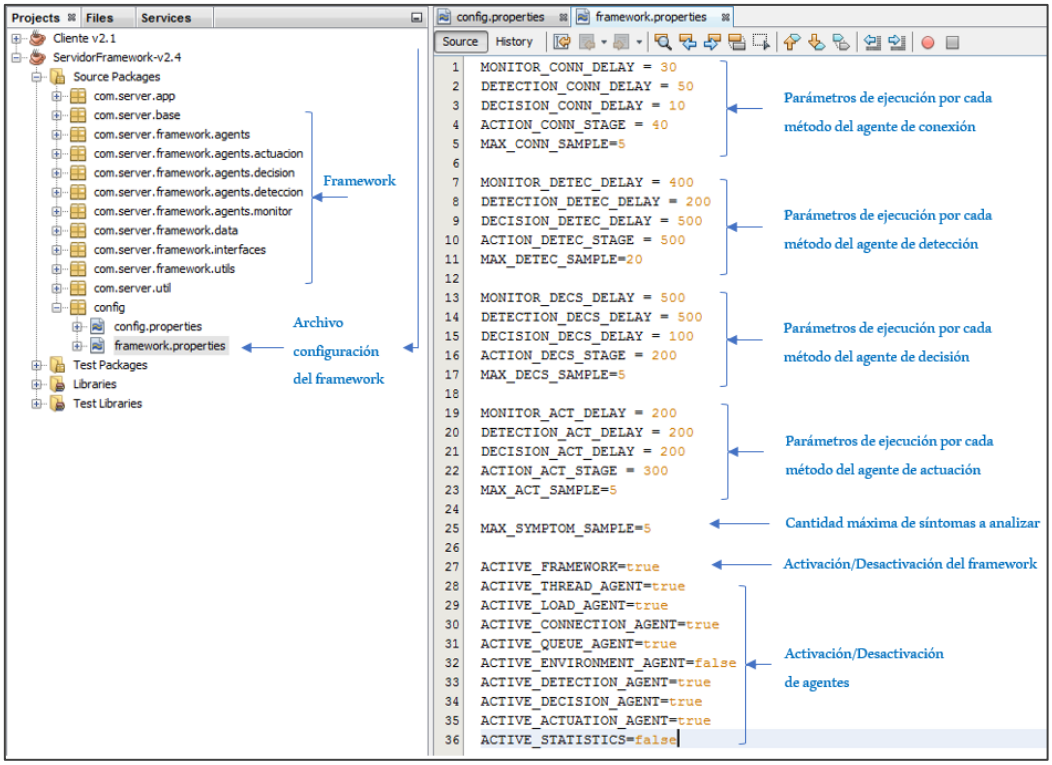
2. Configurar los parámetros del cliente, definiendo cadena de conexión, archivo de Testing a utilizar tiempo de espera entre paquete de solicitudes.



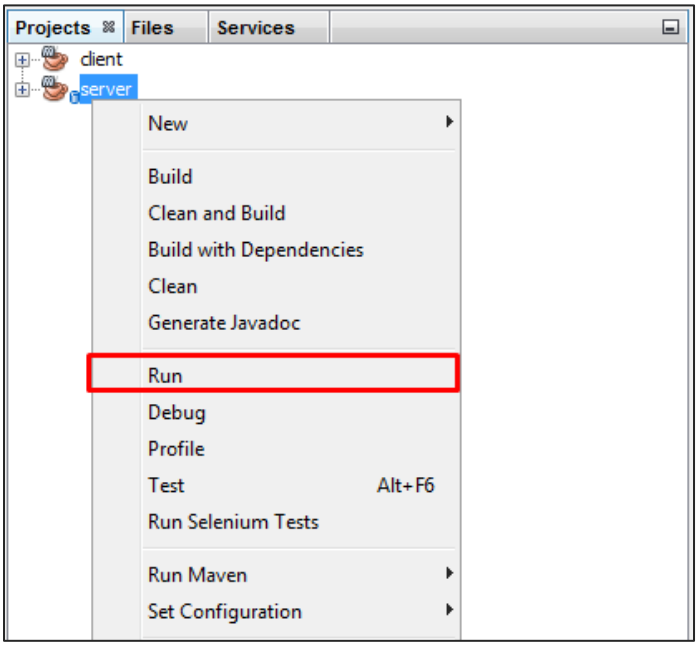
3. Configurar los parámetros del servidor, definiendo cadena de conexión, numero de hilos y colas iniciales.



4. Configurar los parámetros del framework, definiendo cadena de conexión, archivo de Testing a utilizar tiempo de espera entre paquete de solicitudes.



5. Para replicar las pruebas se ejecuta cada módulo iniciando por el servidor y luego el cliente.

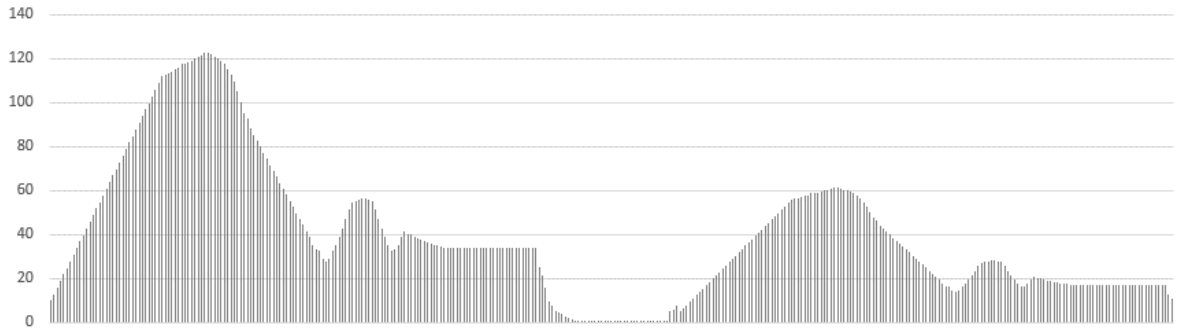


7.1.5 Condiciones externas

Tres son los tipos de demanda que se realizará al aplicativo de prueba, el programa cliente enviará cada cierto periodo de tiempo un grupo completo de solicitudes a una instancia del servidor, la cantidad de solicitudes variará en el tiempo, con esta situación se espera que el marco de trabajo reconozca el volumen de solicitudes encoladas y la demanda de procesamiento, generando una adaptación para responder esta.

Para ilustrar mejor la manera que se llevó a cabo la prueba la *Figura 54* muestra la gráfica de demanda con la distribución del número de peticiones en el tiempo

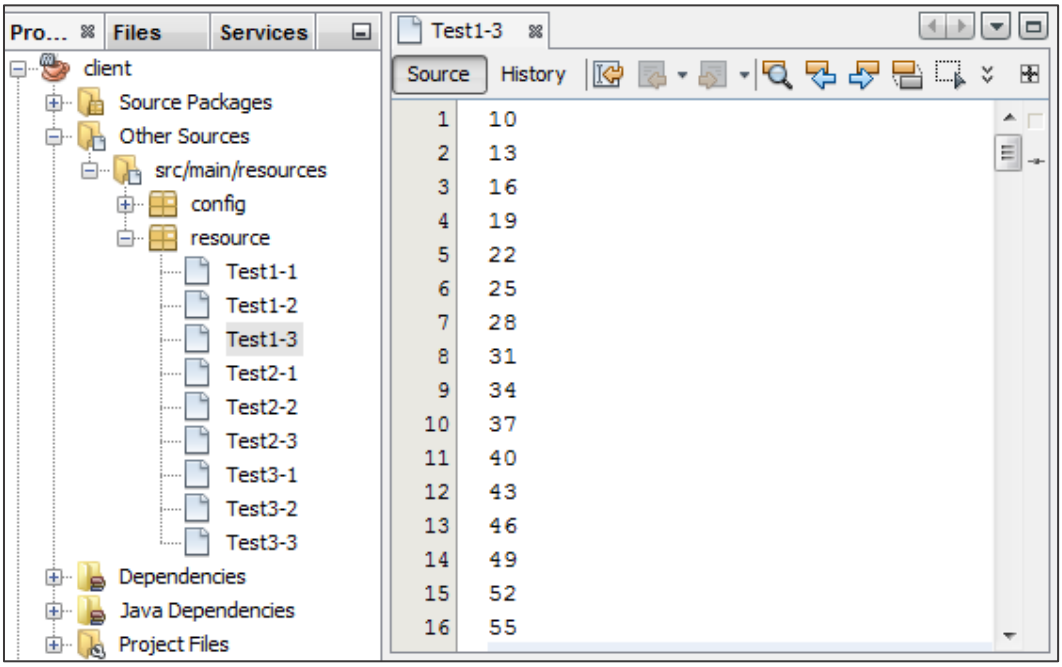
Figura 54. Primera gráfica de demanda muestra el número de peticiones en el tiempo (Entrada 1)



Fuente. Elaboración personal

La imagen anterior se traduce en el marco de trabajo como una secuencia de valores que representan la cantidad de peticiones que la aplicación cliente realizará hacia al servidor cada periodo de tiempo. El archivo que corresponde con la gráfica anterior es *Test-1-3* cuyas primeras líneas se presentan en la *Figura 55* y hacen referencia a los primeros 16 ciclos de envió de peticiones, así como se presenta para el primer instante se envían 10 peticiones, para el siguiente 13 y así sucesivamente.

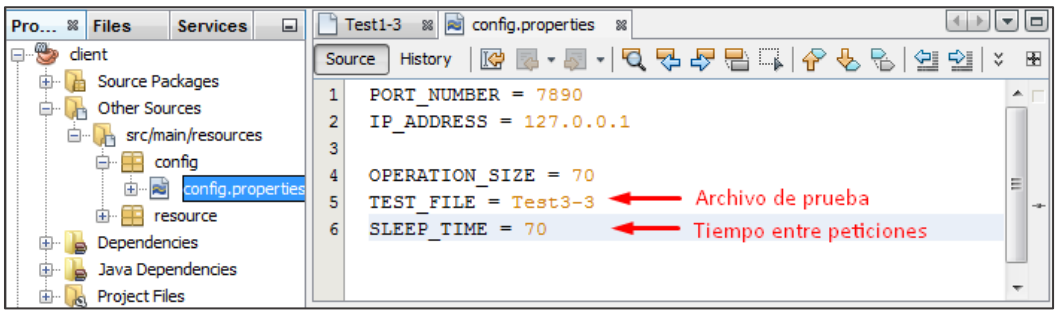
Figura 55. Archivo de entrada de prueba Test1-3



Fuente. Screenshot Netbeans

Para que la aplicación de prueba pueda funcionar utilizando este archivo, se debe acceder a la aplicación client y en ella modificar el archivo de configuración estableciendo el tiempo el periodo de tiempo y el archivo de prueba, tal como se aclara en la *Figura 56*.

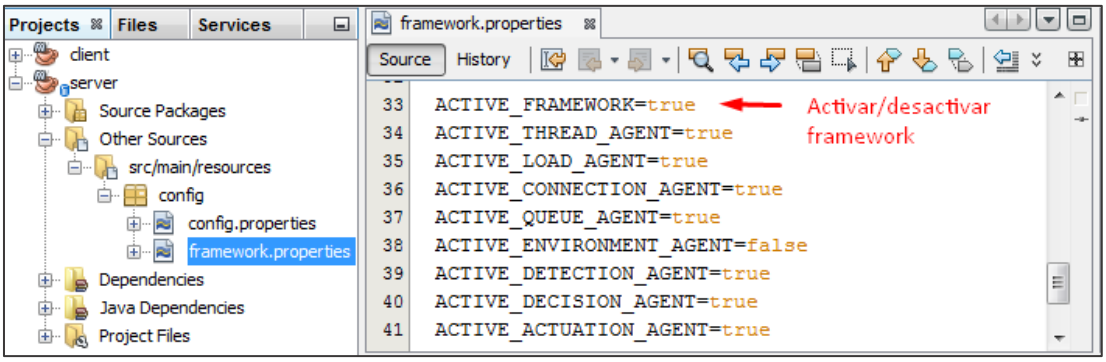
Figura 56. Archivo de configuración



Fuente. Screenshot Netbeans

Cada escenario será sometido a diferentes archivos de entrada, esto es, cada escenario deberá responder a tres diferentes tipos de demanda. Primero, se ejecutará el programa sin aplicar el marco de trabajo y luego se repite el mismo proceso, solo que esta vez con la aplicación del marco del trabajo. La activación o desactivación del marco de trabajo se realiza mediante el archivo de configuración *frameworks.properties* como se indica en la *Figura 57*.

Figura 57. Activación/desactivación del marco de trabajo

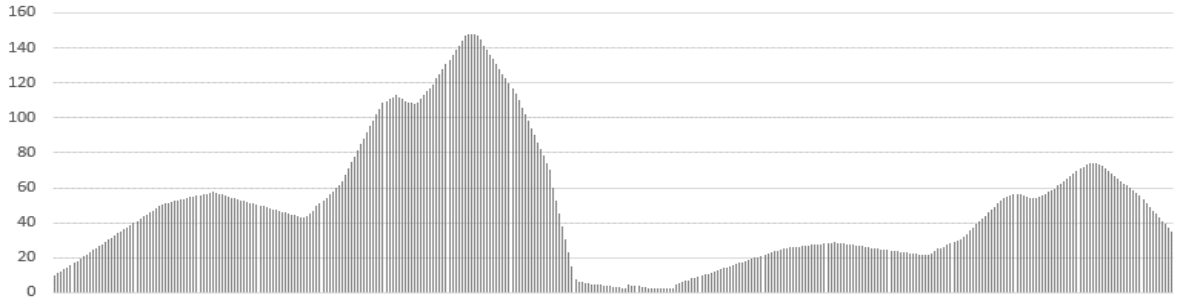


Fuente. Screenshot Netbeans

Para cada entrada de datos se realizará la medición desde el módulo del cliente de los tiempos de respuesta para el paquete de peticiones, estas se encuentran identificadas y secuenciadas de tal manera que faciliten su posterior comparación ambos resultados del mismo tipo de entrada, además se analiza el *estado final* de los elementos que se pueden cambiar para chequear si se presentó o no una adaptación y la manera en que ésta afecta el tiempo de respuesta.

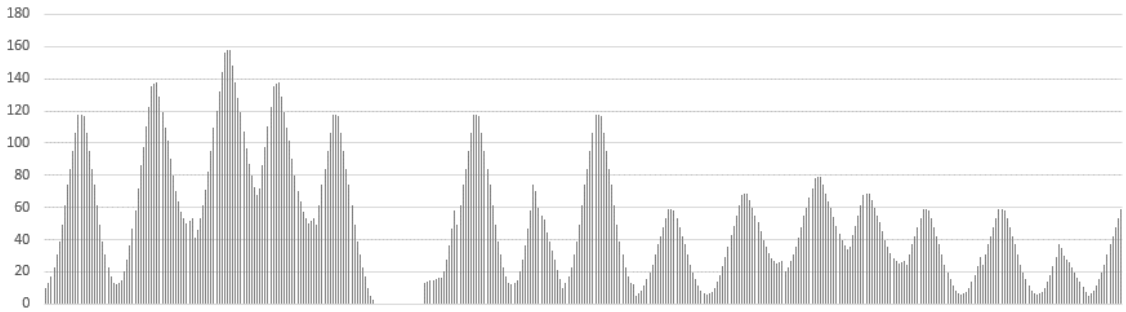
Los otros dos archivos para los dos tipos de demanda mencionados corresponden con los archivos *Test 2-3* y *Test 3-3* cuyas respectivas graficas de demanda se presentan en la *Figura 58* y *Figura 59*.

Figura 58. Segunda gráfica de demanda muestra el número de peticiones en el tiempo (Entrada 2)



Fuente. Elaboración personal

Figura 59. Tercera gráfica de demanda muestra el número de peticiones en el tiempo (Entrada 2)

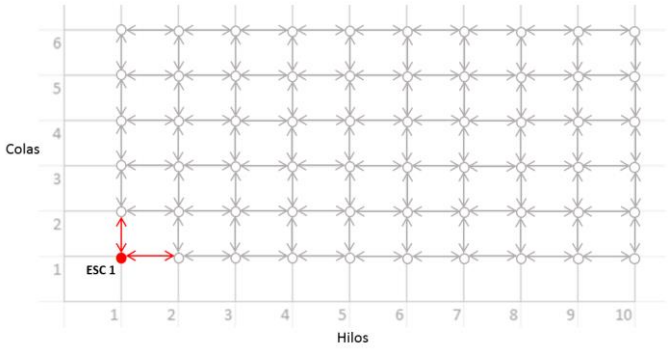


Fuente. Elaboración personal

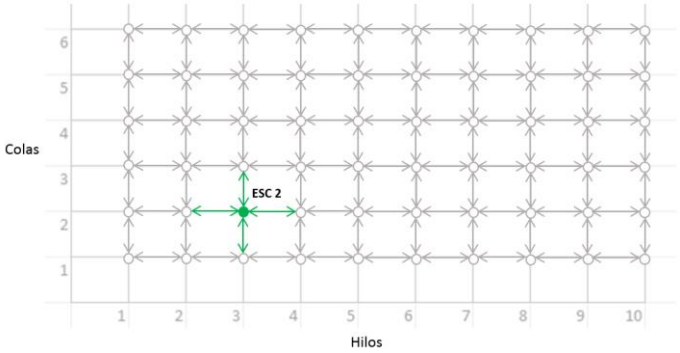
7.1.7 Escenarios de prueba

Para la validación se emplearon los tres escenarios iniciales descritos en detalle a continuación, cada uno de ellos será sometido a cada una de las entradas y al final se analiza el impacto del marco del trabajo tanto en su estado como en la operación del mismo sistema:

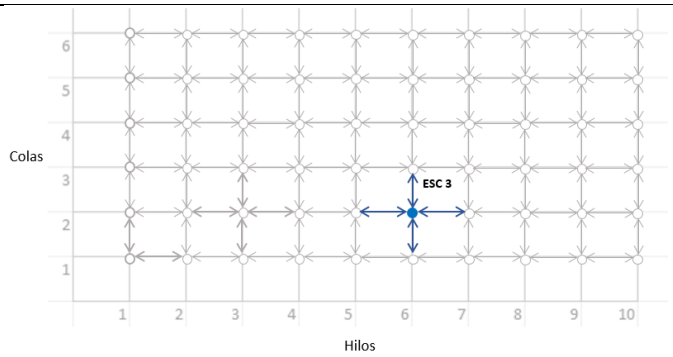
INSTRUMENTO DE REGISTRO			
Nombre	Único hilo/Única cola	**Código**	001

PARAMETROS	
Numero inicial hilos	1
Numero inicial colas	1
Periodo de envío	Cada 70 ms
Frecuencia de monitoreo	2000 ms
Frecuencia de detección	1600 ms
Frecuencia de decisión	1300 ms
Tamaño de operación	Matrices de 70x70
ESCENARIO	
Condiciones internas.	
Descripción	
El estado interno inicial del sistema para este escenario configura el sistema con un una cola de espera y un único hilo trabajador que se encargará de atender todas las peticiones. Estas condiciones iniciales no permiten que la adaptación llegue a desestabilizar el sistema dado que internamente se definieron los valores para los recursos como restricciones de cota inferior. El tamaño de la operación requerida por las instancias *Cliente* se ha establecido a una operación de multiplicación matricial sobre dos matrices de dimensiones equivalentes a 70x70.	

INSTRUMENTO DE REGISTRO			
Nombre	Tres hilos/ Dos colas	**Código**	002
Numero inicial hilos	3		

Numero inicial colas	2
Periodo de envío	Cada 70 ms
Frecuencia de monitoreo	2000 ms
Frecuencia de detección	1600 ms
Frecuencia de decisión	1300 ms
Tamaño de operación	Matrices de 70x70
ESCENARIO	
Condiciones internas.	
Descripción	
El estado interno inicial del sistema para este escenario configura el sistema con un *dos colas* y *tres hilos trabajador* que se encargaran de atender todas las peticiones, la decisión anterior se tomó al considerar las gráficas de tiempos de respuesta para el programa de evaluación de hilos y colas, con esta configuración sin el marco de trabajo y presento esta como uno de los mínimos en tiempos de respuesta según la Figura 51. Estas condiciones iniciales no permiten que la adaptación llegue a desestabilizar el sistema dado que internamente se definieron estos los valores para un punto intermedio con buen rendimiento para los recursos. El tamaño de la operación requerida por las instancias *Cliente* se ha establecido a una operación de multiplicación matricial sobre dos matrices de dimensiones equivalentes a 70x70, generadas aleatoriamente.	

INSTRUMENTO DE REGISTRO

Nombre		**Código**	003
Numero inicial hilos	6		
Numero inicial colas	3		
Periodo de envío	Cada 70 ms		
Frecuencia de monitoreo	2000 ms		
Frecuencia de detección	1600 ms		
Frecuencia de decisión	1300 ms		
Tamaño de operación	Matrices de 70x70		
ESCENARIO			
Condiciones internas.			
Descripción			
El estado interno inicial del sistema para este escenario configura el sistema con un *tres colas* y *seis hilos trabajador* que se encargaran de atender todas las peticiones, la decisión anterior se tomó de considerar las gráficas de tiempos de respuesta para el programa con esta configuración, pero sin el marco de trabajo y al revisar la misma se mostró como uno de los peores escenarios según la Figura 51. Estas condiciones iniciales son desfavorables según las pruebas realizadas además mostró ser uno de los peores escenarios para dar respuesta a las diferentes solicitudes, pero se espera que se presenten microajustes que lo conduzca a una configuración mejor. El tamaño de la operación requerida se ha establecido a una operación de multiplicación matricial sobre dos matrices de dimensiones equivalentes a 70x70, generadas aleatoriamente.			

7.2 ANÁLISIS DE RESULTADOS

La siguiente tabla resume los resultados de la prueba para cada tipo de entrada y escenario, permitiendo la comparación directa entre ellos.

Figura 60. Resultados de evaluación del marco de trabajo

Test-1-3									
	(1, 1) -> (1, 2)			(2, 3) -> (2, 4)			(2, 6) -> (3, 6)		
	Estado inicial	Estado final	Con Framework	Estado inicial	Estado final	Con Framework	Estado inicial	Estado final	Con Framework
Mayor	142491	81355	143708	88829	31282	90486	19155	18084	20060
Menor	381	320	340	463	1757	354	1623	15	2650
Media	73748	44033	75167	35508	18237	33811	9904	7878	10045
Desviación	40413	22829	40239	23621	6834	23896	4698	4854	3313
Adaptación	N/A	N/A	SI	N/A	N/A	SI	N/A	N/A	NO
Estable	SI	SI	SI	SI	SI	SI	SI	SI	SI
	Test-2-3								
	(1, 1) -> (1, 3)			(2, 3) -> (2, 4)			(2, 6) -> (3, 6)		
	Estado inicial	Estado final	Con Framework	Estado inicial	Estado final	Con Framework	Estado inicial	Estado final	Con Framework
Mayor	118326	47518	119418	71152	45755	66444	10493	19587	8883
Menor	274	314	294	700	6790	863	2948	15	2236
Media	61426	24071	62069	28455	25488	26858	6481	8559	3832
Desviación	33210	12131	33574	18529	11216	17533	1609	4342	807
Adaptación	N/A	N/A	SI	N/A	N/A	SI	N/A	N/A	NO
Estable	SI	SI	SI	SI	SI	SI	NO	NO	NO
	Test-3-3								
	(1, 1) -> (1, 4)			(2, 3) -> (1, 3)			(2, 6) -> (2, 7)		
	Estado inicial	Estado final	Con Framework	Estado inicial	Estado final	Con Framework	Estado inicial	Estado final	Con Framework
Mayor	104365	21641	103772	54289	36256	57064	13927	12721	14443
Menor	310	410	294	295	4561	732	47	967	15
Media	54429	12081	54316	22244	21348	23789	5946	5212	7351
Desviación	29248	6301	29086	14177	8276	15208	3456	2414	3724
Adaptación	N/A	N/A	SI	N/A	N/A	NO	N/A	N/A	NO
Estable	SI	SI	SI	SI	SI	SI	SI	SI	SI

Fuente. Elaboración personal

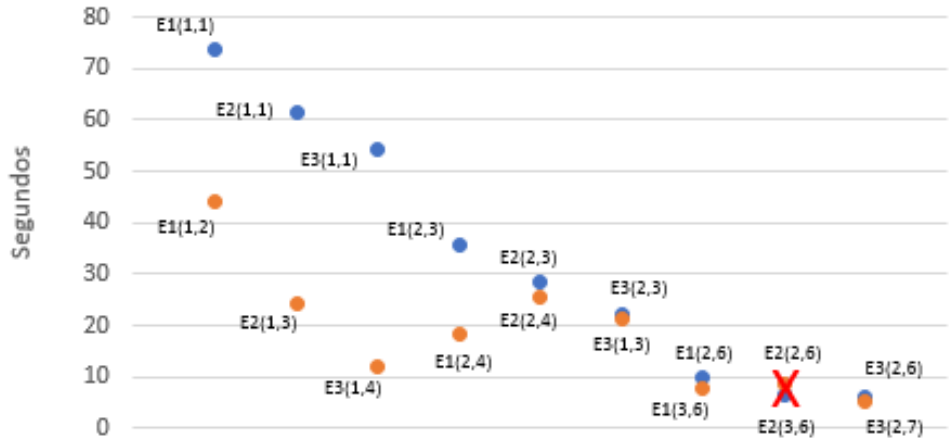
El título de subtabla representa el tipo de entrada que se evalúa (Test-1-3, Test-2-3 y Test-3-3), debajo de este se muestra la transición y estados que se realiza, (2, 3) -> (2, 4) indicando que se consideran el estado inicial (2 colas, 3 hilos), el estado final (2 colas, 4 hilos) y la transición (-> microajuste) entre ellos. De cada una de estas transiciones se toma información para el estado inicial (2,3), esta misma información es registrada para el estado final (2,4), para este ejemplo en particular, se puede notar como el tiempo de atención

requerido cuando se aumenta un hilo, es decir, se pasa de utilizar 3 a utilizar 4 presenta una reducción en el tiempo de respuesta de 60000 ms que representa cerca de un minuto de diferencia. Ahora bien, dada la mejora en estos dos estados se pasa a activar el marco de trabajo y se nota que, aunque se presenta la adaptación, es decir, la transición entre el estado inicial y final (2, 3) -> (2, 4), esto es, se crea un hilo durante extra en tiempo de ejecución, esto no necesariamente impacta positivamente el tiempo de respuesta en comparación con el estado inicial, e incluso para este caso aumenta el tiempo de respuesta cerca de 1 segundo.

Tal como fue descrito, re reportan las salidas y se obtiene la media de atención por cada set de entrada de datos, así, por ejemplo, para el set de la Figura 55, se tienen alrededor de 9000 solicitudes y de ellas se obtiene el tiempo de cada solicitud, del registro de estos datos se extrae: tiempo mayor de respuesta, tiempo menor, tiempo medio y desviación estándar. Para el caso en que se active el marco de trabajo se evalúa si se consigue o no la adaptación, es decir, si se desencadena la transición entre ambos estados provocando el microajuste. Al final, se evalúa la estabilidad de la aplicación *Cliente/Servidor* para todos los escenarios, con y sin marco de trabajo. De las pruebas realizadas se extraen las siguientes conclusiones:

- Comparando los registros de las columnas de estado inicial y final, comparados en la *Figura 61*, y recordando que el marco de trabajo se encuentra *inactivo*, se evidencia que el cambio aplicado evidentemente mejora la respuesta del sistema (*exceptuando el caso de [(2, 6) -> (3, 6)] que presento un estado inestable en la ejecución*), sin embargo, se resalta que la cantidad de elementos (colas de espera e hilos de trabajo) instanciados se mantuvieron en unos rangos aceptables para las pruebas.

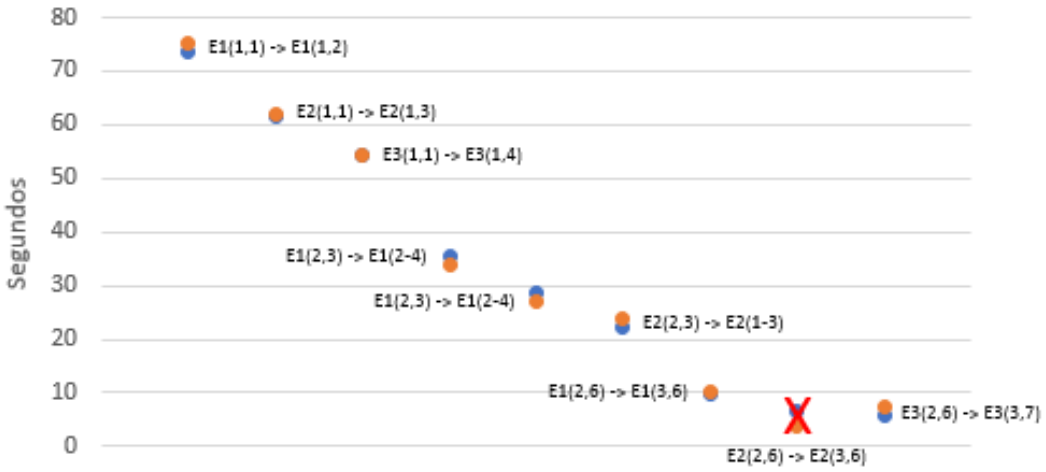
Figura 61. Comparativa entre estado inicial y estado final con marco de trabajo inactivo.



Fuente. Elaboración personal

- Otro aspecto que revela la *Figura 61* es que de observa un impacto en el rendimiento más significativo para aquellas combinaciones donde se varían los hilos trabajadores, más que en aquellos casos en que se varían las colas de espera.
- Al activar el marco de trabajo y comparar los tiempos de respuesta con el tiempo de respuesta para el estado inicial (sin este), se observa una diferencia mínima que oscila entre los (1500 ms y 2600 ms) indicando el valor subrayado un ahorro, mientras el otro un consumo extra de tiempo. De nuevo se excluye del análisis el escenario que se presentó inestabilidad, esta información se muestra en la *Figura 62*.

Figura 62. Comparativa entre marco de trabajo activo e inactivo



Fuente. Elaboración personal

- Los resultados anteriores muestran que la lógica del marco de trabajo activo no afecta significativamente el rendimiento de la aplicación. Es muy importante resaltar que 5 de los 9 casos de prueba realizaron la adaptación programada en tiempo de ejecución. Lo anterior no debe considerarse como que el marco de trabajo no realizó su tarea, de hecho, esto significa que, según las restricciones de operación dadas para el mismo, no se detectó una situación que requiriera la adaptación.
- El escenario que presentó inestabilidad se probó reiteradas veces obteniendo siempre un bloqueo y caídas reiteradas del sistema, a pesar de que se manejaron varias hipótesis sobre la razón de esta situación no se pudo corregir esta situación.

7.3 DISCUSIÓN FINAL

A la pregunta formulada para este trabajo: ¿Es posible adaptar el diseño de un sistema software distribuido, sin perturbar su funcionalidad, mediante la

aplicación de un conjunto de reglas, principios y conocimientos basadas en el estudio de los sistemas adaptativos complejos?, la respuesta es sí, los resultados de las pruebas efectuadas respaldan que es posible adaptar el diseño del software mediante la aplicación del marco de trabajo propuesto para tal fin. Pero también se evidencia, en el desarrollo de este trabajo, que la adaptación requiere de la implementación de mucha lógica de programación y que no mejora los tiempos de respuesta del programa, aun cuando, la adaptación se efectué hacia un estado interno que *sin* la aplicación del marco de trabajo si mejora el tiempo de respuesta como lo muestran los resultados.

En cuanto a la cuestión secundaria de si *en un diseño adaptable es posible efectuar cambios en términos de entidades que asumen roles específicos en el proceso*, esto queda comprobado al revisar cómo fue posible llevar del diseño a la implementación en un lenguaje de programación orientado a objetos las 11 clases diferentes de agentes ver como funcionaron conjuntamente para efectuar la adaptación del sistema entre estados internos. También se comprueba que *las condiciones iniciales del sistema sesgan y limitan los mecanismos de ejecución de la adaptación y es uno de los factores más influyentes al dirigir el sistema hacia la misma*, esto se comprobó al analizar aquellas situaciones que no desencadenaron el microajuste, donde previamente en la descripción del escenario se descrito esta condición. Finalmente, al considerar si *se pueden realizar adaptaciones en tiempo de ejecución sin impactar negativamente el rendimiento del sistema*, los resultados presentan un 89% confiabilidad, dado que uno de los escenarios probados desestabilizó el sistema sin razón aparente.

7.4 CONCLUSIONES DEL PROYECTO

- El marco de trabajo implementado se ha construido teniendo en cuenta las definiciones de sistema adaptativo complejo expuestas en los primeros capítulos, éste permite adaptar el diseño en tiempo de ejecución, los resultados de las pruebas realizadas al caso de estudio descrito en detalle en el capítulo 6, en un ambiente controlado, han demostrado que el sistema acepta microajustes en elementos predefinidos para el diseño. Basado en los resultados obtenidos, ocho (8) de los nueve (9) escenarios trabajaron correctamente, esto se traduce en un 88.9% de efectividad, sin embargo, es importante resaltar que los escenarios exitosos no presentaron una mejora en tiempo de respuesta con el marco de trabajo activo.
- Un sistema basado en agentes ofrece una aproximación viable para incluir la adaptación en un sistema, esto se refleja en los resultados de este proyecto, sin embargo, es necesario tener en cuenta que es una aproximación costosa en términos de horas de desarrollo debido a la gran cantidad de lógica que es necesario programar para lograr el proceso de adaptación como se muestra en el capítulo 4 donde se presenta el diseño del marco de trabajo y donde cada agente debe: 1) encapsular un método de razonamiento específico a su tipo sobre un elemento particular del sistema y 2) proveer los mecanismos de interacción y comunicación con otros agentes, que como fue mencionado son diferentes según las relaciones esperadas con estos.
- Una característica por resaltar del marco de trabajo, es que sin importar el estilo arquitectónico elegido éste desarrolla ideas que pueden ser implementadas independiente de la arquitectura seleccionada, el caso de estudio en el capítulo 6, presenta una arquitectura cliente/servidor donde cada componente se implementa de forma separada y utilizando un paradigma de programación orientado a objetos, una propuesta de trabajo

futura en esta línea seria validar la anterior proposición utilizando alguno de los estilos arquitectónicos expuestos en el capítulo 3.

- Una línea de profundización, para ser desarrollada en trabajos futuros, consiste en utilizar en las etapas de detección y decisión agentes que desplieguen el espacio del discurso de solución, incluyendo todas las posibilidades de micro-ajustes que el esquema original permita, limitado únicamente por sus restricciones en su capacidad de cambio. De esta forma, los agentes de decisión podrían utilizar este espacio de solución para navegar por él y facilitar la toma de decisión hacia configuraciones con mejores características para el sistema.

BIBLIOGRAFIA

GORTON, Ian. Essential Software Architecture. Páginas 271-274, Springer-Verlag Berlin Heidelberg 2006.

TAYLOR, Richard N. y MEDVIDOVIC, Nenad y DASHOFY, Eric M. Software Architecture. Foundations, Theory and Practice. Paginas 1 - 22. Wiley, 2010

BUDGEN, David. Software design. Pearson education Limited. Segunda edición, 2003.

QIAN Kai, FU Xian, TAO Lixin, XU Chong-Wei, Jorge L.Díaz-Herrera. Software Architecture and Design Illuminated. Jones and Bartlett, 2010.

KOSSIAKOFF Alexander, SWEET William N., SEYMOUR Samuel J., BIEMER Steven M. System Engineering Principles and Practice. Wiley, 2010.

ROZANSKI Nick, WOODS Eoin. Software System Architecture. Addison Wesley, 2012.

COULOURIS George, DOLLIMORE Jean, KINDBERG Tim. Sistemas distribuidos. Conceptos y diseño. Addison Wesley, 2001.

CLEMENTS Paul, KAZMAN Rick, KLEIN Mark. Evaluating Software Architecture. Methods and Case Studies. Addison Wesley, 2011.

FREEMAN Eric, FREEMAN Elisabeth, SIERRA Kathy, BATES Bert. Head First Design Patterns. O'Reilly Media, Inc, 2004.

AURUM Aybüke, WOHLIN Claes. Engineering and Managing Software Requirements. Springer Science & Business Media. 2006

KEPHART, Jeffrey O. & WALSH, William E. “An Artificial Intelligence Perspective on Autonomic Computing Policies,” 3-12. Proceedings of the Fifth IEEE International Workshop on Policies for Distributed Systems and Networks (Policy 2004). Yorktown Heights, NY, June 7-9, 2004. Los Alamitos, CA: IEEE Computer Society, 2004.

MÜLLER, Hausi A., et al. *Autonomic computing*. CARNEGIE-MELLON UNIV PITTSBURGH PA SOFTWARE ENGINEERING INST, 2006.

WHITE Steve R., HANSON James E., WHALLEY Ian, CHESS David M., y Jeffrey O.Kephart. An architectural approach to Autonomic computing. IBM Thomas J.Watson Research Center. 2004.

LAPOUCHNIAN, Alexei y LIASKOS, Sotirios y MYLOPOULOS, Jhon y YU Yijun. Towards Requirements-Driven Autonomic System Design. Department of Computer Science University of Toronto. 2005.

SALEHIE, Mazeiar; TAHVILDARI, Ladan. Autonomic computing: emerging trends and open problems. En *ACM SIGSOFT Software Engineering Notes*. ACM, 2005. p. 1-7.

HUEBSCHER, Markus y McCANN, Julie A. Evaluation issues in autonomic computing. Department of computing, Imperial College London. 2003.

COMPUTING, Autonomic. An architectural blueprint for autonomic computing. *IBM Publication*, 2003.

HUEBSCHER, Markus C y MCCANN, Julie A. A survey of Autonomic Computing — degrees, models and applications. Imperial College London. ACM Computing Surveys (CSUR), 2008

SALEHIE, Mazeiar; TAHVILDARI, Ladan. Self-adaptive software: Landscape and research challenges. *ACM Transactions on Autonomous and Adaptive Systems (TAAS)*, 2009, vol. 4, no 2, p. 14.

WHITE, Steve R. HANSON, James E. IAN WHALLEY, CHESS, David M. and JEFFREY O.Kephart. An architectural approach to Automotic computing. IBM Thomas J.Watson Research Center. 2004.

BLAIR G.S, COULSON G., Robin P, PAPATHOMAS M. An architecture for next generation middleware. Distributing Multimedia Research Group. Computing Department, Lancaster University. 2000.

MEDVIDOVIC, Nenad. ADLs and Dynamic Architecture Changes. Department of Information and Computer Science. 2000.

MEDVIDOVIC, Nenad. OREIZY Peyman, TAYLOR Richard N. Architecture-Based Runtime Software Evolution Department of Information and Computer Science. University of California. 2000.

TAYLOR, Richard N. Nenad Medvidovic, Eric M. Dashofy. Software Architecture. Foundations, Theory and Practice. Wiley, 2010

MORIN, Brice, et al. Unifying runtime adaptation and design evolution. En *Computer and Information Technology, 2009. CIT'09. Ninth IEEE International Conference on*. IEEE, 2009. p. 104-109.

GEIHS, Kurt. Selbst-adaptive software. *Informatik-Spektrum*, 2008, vol. 31, no 2, p. 133-145.

KEPHART, Jeffrey O.; CHESS, David M. The vision of autonomic computing. *Computer*, 2003, vol. 36, no 1, p. 41-50.

BRANKE, Jürgen, et al. Organic Computing–Addressing complexity by controlled self-organization. En *Leveraging Applications of Formal Methods, Verification and Validation, 2006. ISoLA 2006. Second International Symposium on*. IEEE, 2006. p. 185-191.

PREISLER, T., & RENZ, W. Structural Adaptations for Self-Organizing Multi-Agent Systems.

WOOLRIDGE. M. Multi-Agent Systems: An Introduction. John Wiley & Sons (Chichester, England), 2001.

WEYNS, Danny and GEORGEFF, Michael. Self-Adaptation Using Multiagent Systems. Software technology. IEEE Software 2010.

VOKOVIK, M.. Context Aware Service Composition. PhD thesis, University of Cambridge, 2006.

ŚNIEŻYŃSKI, Bartłomiej. Agent-based Adaptation System for Service-Oriented Architectures Using Supervised Learning. Procedia Computer Science. Volume 29, 2014, Pages 1057–1067

ŚNIEŻYŃSKI, Bartłomiej and DAJDA, Jacek. Comparison of strategy learning methods in farmerpest problem for various complexity environments without delays. Journal of Computational Science,
4(3):144 – 151, 2013.

ŚNIEŻYŃSKI, Bartłomiej. Agent strategy generation by rule induction. Computing and Informatics, 32(5):1055–1078, 2013.

QURESHI, N. A., & PERINI, A. (2008, August). An agent-based middleware for adaptive systems. In 2008 The Eighth International Conference on Quality Software (pp. 423-428). IEEE.

GUANG, Liang; PLOSILA, Juha; TENHUNEN, Hannu. From self-aware building blocks to self-organizing systems with hierarchical agent-based adaptation. En *Proceedings of the 2014 International Conference on Hardware/Software Codesign and System Synthesis*. ACM, 2014. p. 23.

PAN, Jiali; HAN, Minghong. The evolutionary optimization of system of systems based on Agent model. En *2014 IEEE International Conference on System Science and Engineering (ICSSE)*. IEEE, 2014. p. 231-235.

JIAO, Wenpin; SUN, Yanchun. Self-adaptation of multi-agent systems in dynamic environments based on experience exchanges. *Journal of Systems and Software*, 2016, vol. 122, p. 165-179.

GELL-MANN, Murray. El quark y el jaguar. Aventuras en lo simple y en lo complejo. Libros para pensar la ciencia. Colección dirigida por Jorge Wagensberg. Metatemas, 1994.

FLAKE, Halvar. Structural comparison of executable objects. 2004.